Towards an open digital audio workstation for live performance

the development of an open soundcard

Dimitrov, Smilen

DOI (link to publication from Publisher):
10.5278/vbn.phd.engsci.00028

Publication date:
2015

Document Version
Publisher's PDF, also known as Version of record

TOWARDS AN OPEN DIGITAL AUDIO WORKSTATION FOR LIVE PERFORMANCE: THE DEVELOPMENT OF AN OPEN SOUNDCARD

BY
SMILEN DIMITROV

DISSERTATION SUBMITTED 2015

AALBORG UNIVERSITY
DENMARK

Towards an open digital audio workstation for live performance: the development of an open soundcard

Ph.D. Dissertation

Smilen Dimitrov

Dissertation submitted June 18, 2015

Thesis submitted: June 18, 2015
PhD Supervisor: Prof. Stefania Serafin
Aalborg University
PhD Committee: Assoc. Prof. Olga Timčenko, Aalborg University
Prof. Anna Friesel, Technical University of Denmark (DTU)
Prof. Nicola Bernardini, Conservatorio di Musica "Santa Cecilia"
PhD Series: Faculty of Engineering and Science, Aalborg University

Dimitrov, Smilen. "Towards an open digital audio workstation for live performance: the development of an open soundcard" / "På vej mod en åben, digital lyd-arbejdsstation til live-optræden: Udviklingen af et åbent lydkort"

ISSN (online): 2246-1248
ISBN (online): 978-87-7112-311-1

Published by:
Aalborg University Press
Skjernvej 4A, 2nd floor
DK – 9220 Aalborg Ø
Phone: +45 99407140
aauf@forlag.aau.dk
forlag.aau.dk

Printed in Denmark by Rosendahls, 2015

This document was typeset with pdfLaTeX (pdfTeX, Version 3.14159265-2.6-1.40.15 (TeX Live 2014)), based on template aau_phd_thesis_template_v1.2.1.zip, document class book, using packages: iftex, hologo, setspace, trace, lipsum, cmap, fontenc, babel, textcomp, inputenc, lmodern, mathpazo, paratype, tgpagella, substitutefont, microtype, etoolbox, xcolor, graphicx, caption, array, booktabs, tabularx, multirow, framed, tikz, pgfplots, tikzpagenodes, lilyglyphs, ifluatex, ifxetex, amsmath, amssymb, ntheorem, geometry, titlesec, fancyhdr, calc, mparhack, csquotes, biblatex, xpatch, tocbibind, appendix, lastpage, todonotes, soulutf8, pdftexcmds, hyperref, adjustbox, nth, pdfpages, pax, glossaries, units, siunitx, hhline, comment, listings, grffile, fixfoot, afterpage, CJKutf8, multicol, mathtools, environ, xstring, filecontents, pgfplotstable, tikz-timing, rotating. Additional open-source software (such as gnuplot, numpy, matplotlib, pdftk, qpdf, Ghostscript, ImageMagick, GIMP, Inkscape, and others) has been used in preparation of data and figures in this document.

Curriculum Vitae

Smilen Dimitrov

Born in 1977, in Skopje, Macedonia; nationality Macedonian. Diverse areas of interest and involvement, mainly revolving around music and technology; some career development highlights are summarized below.

Education

1983–1988	–, *M.B.U.C "Ilija Nikolovski - Luj"*, Skopje, Socialist Republic of Macedonia, Socialist Federal Republic of Yugoslavia. Attended (and dropped out of) elementary music school: music theory (solfège) and piano
1984–1992	–, *O.U. "Johann Heinrich Pestalozzi"*, Skopje, Socialist Republic of Macedonia, Socialist Federal Republic of Yugoslavia. Attended elementary school
1992–1994	–, *S.U. "Orce Nikolov"*, Skopje, Republic of Macedonia. Attended high school, specializing in electronics
1994–1995	**High School Diploma**, *"Bonny Eagle" H. S.*, Standish, Maine, United States of America. Attended high school, acquired high-school diploma
1995–2001	**B.Sc.**, *Faculty of Electrotechnical engineering, University "Sts. Cyril & Methodius"*, Skopje, Republic of Macedonia. Obtained Bachelor of Science in Electronics & Telecommunication

2002–2004 **Multimedia Designer**, *Aarhus Technical College*, Århus, Denmark.
Studied web design; trainee service in Brother, Brother & Sons ApS, Copenhagen (software for 3D visualization of sensor-equipped machine parts)

2004–2006 **M.Sc.**, *Aalborg University*, Copenhagen, Denmark.
Obtained Master of Sciences in Media Technology; master project at Brother, Brother & Sons ApS, Copenhagen (software for 3D visualization, sequencing and control of stage lights)

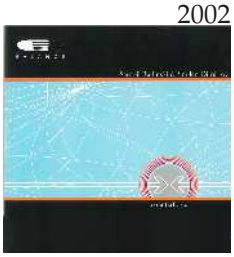
2007–2015 **Ph.D.**, *Aalborg University*, Copenhagen, Denmark.
Studies of media technology; focus on sensors technology, data acquisition and analog/digital interfacing, both in research and as lecturer

Music releases

2000 **S.A.F.**, *Safizam*.
(Profundus - Skopje, Republic of Macedonia)
(Lithium Records - Skopje, Republic of Macedonia)
(Zort Produkcija - Skopje, Republic of Macedonia)
Long-play CD (first release), vinyl (2002 re-release), enhanced CD (2007 re-release), vinyl & CD & compact cassette tape (2015 re-release); as rapper, co-producer

2002 **Pece Atanasovski Orchestra**, *s/t*.
(Amanet Music - Skopje, Republic of Macedonia)
Long-play CD; as *tambura* player

2002 **Samoil Radinski & Smilen Dimitrov**, *Counterforce*.
(Balance - Skopje, Republic of Macedonia)
Single, vinyl; as co-producer



Abstract

The recent, decades-long, successes of electronic music in popular music business, may warrant further inquiries into electronic music instruments. This thesis explores various aspects in this domain, perceived as a part of media technology research. Starting from a formal specification of a problem in the electronic music instrument development domain: how to allow real-time rendering and control, of the playback speed of digital audio loops – a major stumbling block is identified in the ability to design and implement digital audio algorithms in hardware.

Finding that cheap and freely accessible projects, that would serve as practical exercises into issues found in digital audio hardware design and implementation, were near impossible to discover – this thesis responds by identifying the soundcard as suitable for exploration as a generic device, that implements digital audio in both hardware, and computer software. In a series of projects, free & open-source software and documentation was released, which describes various hardware and software aspects of implementing soundcard systems with differing qualities; whose relatively low cost makes them suitable for re-implementation as exercises. These development efforts are documented in peer-reviewed articles, that are collected in part II of this thesis.

An open approach to the design of the soundcard as a computer system device, brings about the conclusion that relatively simple hacks would lead to more versatile and open digital audio platforms; the peer-reviewed articles in part III represent a sample of wider electronic music instrument and media technology research, which might particularly benefit from such developments. However, while free & open-source approaches have clear benefits – and, in fact, may have made projects of this scope possible – in some respects, they are susceptible to the same pressures as proprietary software: most notably, rapid obsolescence. Such challenges are touched upon by this thesis along with the details of more technical nature, as it is likely that they will be unavoidable for any developer, aiming to work with digital audio hardware that can interface with ever-evolving contemporary consumer technology.

Dansk Resumé

De seneste succeser (igennem årtier) af elektronisk musik i det populære musikbranche, sikrer muligvis fortsatte undersøgelser i elektroniske musikinstrumenter. Afhandlingen udforsker forskellige aspekter i dette domæne, set som en del af medieteknologisk forskning. Med udgangspunkt i en formel specifikation af et problem, fra elektronisk musikinstrument udvikling feltet: hvordan at tillade rendering og kontrol i realtid, af afspilningshastigheden af digitale lydløkker - en større anstødssten er identificeret i evnen at designe, samt implementere, digitale lyd algoritmer i hardware.

Eftersom billige og frit tilgængelige projekter, der kunne tjene som praktiske øvelser i problemstillinger fundet i design og implementering af digital lyd hardware, var næsten umulige at spore – denne afhandling svarer med at fastslå lydkortet som egnet til forskning som en generisk enhed, der implementerer digital lyd i både hardware, og computer software. I en projektserie, fri og åbenkilde kode og dokumentation var udgivet, som beskriver forskellige hardware og software aspekter i implementeringen af lydkort systemmer med diverse kvaliteter; hvis relativt lav pris gør dem egnede til genopbygning som øvelser. Disse udviklingsindsatser er dokumenteret i ekspertevaluerede artikler, som er samlet i del II af denne afhandling.

En åben tilgang til designet af an lydkort som en enhed i et computer system, fremhæver konklusionen at relativt enkle modifikationer kunne føre til mere alsidige og åbne digital lyd platformer; peergruppeevaluerede artikler i del III repræsenterer et udsnit fra det bredere forskning i elektroniske musikinstrumenter og medieteknologi, som er et område der kan få særligt gavn af sådanne udviklinger. Imidlertid, mens fri og åben-kilde tilgange har klare fordele – og har muligvis gjort projekter med denne rækkevidde mulige – i visse henseender, er de lige så udsatte over for pres som proprietær softwareudvikling: nemlig hurtig forældelse. Sådanne udfordringer er berørt af denne afhandling sammen med detaljer af mere teknisk natur, da de sandsynligvis vil være uundgåelige for enhver udvikler, som er målrettet til arbejde med digital lyd hardware der kan forbindes med en evigt-udviklende moderne forbruger-teknologi.

Contents

Thesis Details

Thesis Title:	Towards an open digital audio workstation for live performance: the development of an open soundcard
Ph.D. Student:	Smilen Dimitrov
Supervisor:	Prof. Stefania Serafin, Aalborg University Copenhagen

The main body of this thesis consist of the following papers.
Papers in part II, "Papers on open soundcard development":

[II-A] Smilen Dimitrov, "Extending the soundcard for use with generic DC sensors", in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2010)*, Sydney, Australia, Jun. 2010, pp. 303–308, ISSN: 2220-4792, ISBN: 978-0-646-53482-4. URL: `http://imi.aau.dk/~sd/phd/index.php?title=ExtendingISASoundcard`

[II-B] Smilen Dimitrov and Stefania Serafin, "Minivosc - a minimal virtual oscillator driver for ALSA (Advanced Linux Sound Architecture)", in *Proceedings of the Linux Audio Conference (LAC 2012)*, Stanford, California, USA, Apr. 2012, pp. 175–182, ISBN: 978-1-105-62546-6. URL: `http://imi.aau.dk/~sd/phd/index.php?title=Minivosc`

[II-C] Smilen Dimitrov and Stefania Serafin, "Audio Arduino - an ALSA (Advanced Linux Sound Architecture) audio driver for FTDI-based Arduinos", in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2011)*, Oslo, Norway, May 2011, pp. 211–216, ISSN: 2220-4792, ISBN: 978-82-991841-7-5. URL: `http://imi.aau.dk/~sd/phd/index.php?title=AudioArduino`

[II-D] Smilen Dimitrov and Stefania Serafin, "An analog I/O interface board for Audio Arduino open soundcard system", in *Proceedings of the 8th Sound and Music Computing Conference (SMC 2011)*, Padova, Italy: Padova University Press, Jul. 2011, pp. 290–297, ISBN: 978-8-897-38503-5. URL: `http://imi.aau.dk/~sd/phd/index.php?title=AudioArduino-AnalogBoard`

[II-E] Smilen Dimitrov and Stefania Serafin, "Towards an open sound card — a bare-bones FPGA board in context of PC-based digital audio", in

Proceedings of Audio Mostly 2011 - 6th Conference on Interaction with Sound, Coimbra, Portugal, Sep. 2011, pp. 47–54, ISBN: 978-1-4503-1081-9. DOI: 10.1145/2095667.2095674. URL: http://imi.aau.dk/~sd/phd/index.php?title=AudioBareBonesFPGA

[II-F] Smilen Dimitrov and Stefania Serafin, "Open soundcard as a platform for practical, laboratory study of digital audio: a proposal", *International Journal of Innovation and Learning*, vol. 15, no. 1, pp. 1–27, Jan. 2014, ISSN: 1471-8197. DOI: 10.1504/IJIL.2014.058865

[II-G] Smilen Dimitrov and Stefania Serafin, "Comparing the CD-quality, full-duplex timing behavior of a virtual (dummy), hda-intel, and FTDI-based AudioArduino soundcard drivers for Advanced Linux Sound Architecture", *Linux Journal*, 2015, Manuscript submitted/in review

Papers in part III, "Papers on related media technology research":

[III-H] Smilen Dimitrov and Stefania Serafin, "A simple practical approach to a wireless data acquisition board", in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2006)*, IRCAM — Centre Pompidou, Paris, France, Jun. 2006, pp. 184–187, ISSN: 2220-4792, ISBN: 978-2-84426-314-8

[III-I] Kjetil Falkenberg Hansen, Marcos Alonso, and Smilen Dimitrov, "Combining DJ Scratching, Tangible Interfaces And A Physics-Based Model Of Friction Sounds", in *Proceedings of the 2007 International Computer Music Conference (ICMC 2007)*, vol. 2, Copenhagen, Denmark: The International Computer Music Association, Aug. 2007, pp. 45–48, ISBN: 0-9713192-5-1

[III-J] Smilen Dimitrov, Marcos Alonso, and Stefania Serafin, "Developing block-movement, physical-model based objects for the Reactable", in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2008)*, Genova, Italy, Jun. 2008, pp. 211–214, ISBN: 978-88-901344-6-3

[III-K] Luca Turchet, Rolf Nordahl, Stefania Serafin, Amir Berrezag, Smilen Dimitrov, and Vincent Hayward, "Audio-haptic physically-based simulation of walking on different grounds", in *Proceedings IEEE Multimedia Signal Processing Conference (MMSP'10)*, Stéphane Pateux, Ed. Saint Malo, France: IEEE Press, 2010, pp. 269–273, ISBN: 978-1-4244-8110-1. DOI: 10.1109/MMSP.2010.5662031

[III-L] Rolf Nordahl, Amir Berrezag, Smilen Dimitrov, Luca Turchet, Vincent Hayward, and Stefania Serafin, "Preliminary Experiment Combining Virtual Reality Haptic Shoes and Audio Synthesis", *Lecture Notes in Computer Science*, vol. 6192, pp. 123–129, 2010, ISSN: 0302-9743. DOI: 10.1007/978-3-642-14075-4_18

[III-M] Stefania Serafin, Luca Turchet, Rolf Nordahl, Smilen Dimitrov, Amir Berrezag, and Vincent Hayward, "Identification of virtual grounds using virtual reality haptic shoes and sound synthesis", in *Proceedings of the Eurohaptics 2010 Special Symposium*, A. Nijholt, E. O. Dijk, P. M.C. Lemmens, and S. Luitjens, Eds., 1st ed. Ser. CTIT Proceedings WP10-01. Amsterdam, Netherlands: University of Twente, Jul. 2010, pp. 61–70, ISSN: 0929-0672

In addition to the main papers, the following publications by the author have also been made, not included this thesis.

[1] Stefania Serafin, Smilen Dimitrov, Steven Gelineck, Rolf Nordahl, and Olga Timcenko, "Sonic interaction design : case studies from the Medialogy education", in *Proceedings of Audio Mostly 2007 - 2nd Conference on Interaction with Sound*, 2007

[2] Smilen Dimitrov, "Scientific Report from ConGAS Short Term Scientific Mission (STSM) to Stockholm", ConGAS Cost action 287, Tech. Rep., Mar. 2007

[3] Smilen Dimitrov, "Scientific Report from ConGAS Short Term Scientific Mission (STSM) to Barcelona", COST SID IC0601 Action, Tech. Rep., Jan. 2008

[4] Niels Böttcher and Smilen Dimitrov, "An early prototype of the augmented PsychoPhone", in *NIME'09: Proceedings of the 9th Conference on New Interfaces for Musical Expression*, 2009

[5] Luca Turchet, Stefania Serafin, Smilen Dimitrov, and Rolf Nordahl, "Conflicting Audio-haptic Feedback in Physically Based Simulation of Walking Sounds", English, in *Haptic and Audio Interaction Design*, ser. Lecture Notes in Computer Science, Rolf Nordahl, Stefania Serafin, Federico Fontana, and Stephen Brewster, Eds., vol. 6306, Springer Berlin Heidelberg, 2010, pp. 97–106, ISBN: 978-3-642-15840-7. DOI: 10.1007/978-3-642-15841-4_11

[6] Luca Turchet, Stefania Serafin, Smilen Dimitrov, and Rolf Nordahl, "Physically Based Sound Synthesis and Control of Footsteps Sounds", in *Proceedings of the 13th International Conference on Digital Audio Effects (DAFx-10)*, Alois Sontacchi, Hannes Pomberger, and Frans Zotter, Eds., 1st ed. 2010, vol. 1, pp. 161–168, ISBN: 978-3-200-01940-9

This thesis has been submitted for assessment in partial fulfillment of the PhD degree. The thesis is based on the submitted or published scientific papers which are listed above. Parts of the papers are used directly or indirectly in the extended summary of the thesis. As part of the assessment, co-author statements have been made available to the assessment committee and are also available at the Faculty. The thesis is not in its present form acceptable for open publication but only in limited and closed circulation as copyright may not be ensured.

Посветено на моите родители, Владимир и Тодорка Димитрови.

Dedicated to my parents, Vladimir and Todorka Dimitrovi.

Preface

This Ph.D. dissertation is a result of more than seven years of work, that I had done at what is now the Department of Architecture Design and Media Technology, in Aalborg University (AAU) in Copenhagen. During the Ph.D. study period my research work, most of which is documented in this thesis, was tightly coupled with my work as a lecturer in the Medialogy education, led by the department – where I dealt with topics in introductory electronics in undergraduate courses named Sensors Technology, or (later) Physical Interface Design.

The dissertation discusses topics generally within the cross-disciplinary intersection of the areas of music, and electronics and software engineering; interests in which I have been invested for the most of my life. Thus there have been truly many people, to whom I owe thanks as influences on this project (some of which I have attempted to capture in the Acknowledgments further on). However, specifically for my experience during my Ph.D. program at AAU – beyond the support (financial and otherwise) of my parents and my wider family – I am forever thankful to the efforts of the following individuals in providing me with opportunities and support: the Chairman of the Study Board for Media Technology, Dr. Rolf Nordahl; my main thesis supervisor and Professor, Dr. Stefania Serafin; Professor Emeritus, Dr. Erik Granum; and Professor, Dr. Lise Busk Kofoed. These opportunities - beyond conference attendances to places like Sydney, Australia and Genova, Italy - also included two short-term scientific missions (STSMs) to Stockholm, Sweden and Barcelona, Spain, funded by the EU Cooperation in Science and Technology (COST).

This thesis consists of a collection of papers in two parts, for which a detailed summary is provided in an introductory part. I therefore owe my sincere acknowledgment to all co-authors, with whom I've collaborated on publications during this period. The study generally focuses on open-source implementations for digital audio computer hardware; my claim for applying for the Ph.D. degree is that in this period, reducing the available information (even if open) to a format suitable as student exercises, required an amount of research labor commensurate to the academic requirements.

Smilen Dimitrov
Aalborg University, June 18, 2015

Part I

Introduction

Chapter 1

Background

In the middle of the second decade of the 21st century, it is difficult to speak generally of electronic music as something novel or new; and the same can be said of electronic music instruments. Since the origins of the Theremin [1] or Rhythmicon [2] in the 1920s to 1930s, not only are there already decades of academic research in existence on this topic at present, but there are also decades of its commercial influence within popular music. Ignoring the fact that already in the late 20th century, the default understanding of a rock'n'roll setup involved electric guitars, distortion pedals and amplifiers (i.e., *electronic* instruments) – it is still possible to perceive decades-long presence of "genuinely" electronic music on the pop scene: from the historic successes of artists like Jean Michel Jarre and Kraftwerk, to the current success of genres that could be broadly categorized under either hip hop or electronic dance/club music (EDM); in addition to its presence on experimental/subculture/underground scenes, through genres such as industrial or electronic body music (EBM).

A possible cause for the success of the aforementioned artists and genres, could be their ability to inspire joy or meaning in their audiences during their concerts (live performances) – even if the performers use devices, that traditionally wouldn't even be recognized as musical instruments. Technology-wise, the intimate connection of electronic music performers and the recording industry, can be observed in the choices of many artists, to include what would usually be regarded as studio or recording equipment in their live setups. There are differences, however: according to one taxonomy [3], we could categorize electronic music instruments - as controllers - in: (a) instrument-like controllers, (b) extended controllers, and (c) alternative controllers. Thus, Kraftwerk and Jean Michel Jarre would represents users of (generally) acoustic instrument-like interfaces that drive electronic sound generation; while hip hop and EDM genres heavily rely on the presence of a DJ (disk-jockey), whose main instrument is the combination of two turntables and a mixer - neither of which are devices originally designed for a musical instrument purpose (and as such, can be

considered alternative controllers).

The original ambition of this thesis was in the domain of *digital lutherie* (as promulgated by S. Jordà in his thesis [4]), because it starts with identifying specific live performance styles, common in urban varieties of popular electronic music – and responding to that with a design of a digital audio workstation (DAW), that could be used to support those playing styles. In doing so, it would have positioned itself in a decades-long tradition: academically exemplified at the very least by the output of conferences, such as New Interfaces for Musical Expression (NIME), International Computer Music Conference (ICMC) or Digital Audio Effects (DAFX); but also embodied in the work of artists, producers, instrument builders, and researchers.

Specifically in this case, what is common to the musical (and live performance) styles of interest, is their reliance on *audio samples*, considered as (short or long) snippets of audio recordings. In fact, some of these genres can be entirely based on *digital* audio samples, as afforded by their reproduction through instruments such as *samplers* (here, a colloquial synonym for digital audio sampler). The underlying musical research motivation thus lies in exploring an apparent paradox: if an audio sample is an implicitly unchanging recording of past events - how is any ambition towards live, real-time, performance based on samples, any different from the act of mere initiation of automatic playback of an audio recording? While one general answer would be to emulate acoustic musical instruments, the interaction methods that evolved in the genres of interest - with devices not originally designed as musical instruments - can be considered distinct from the acts of playing traditional wind, strings, and percussion musical instruments.

However, addressing issues in this domain often relies, not so much on the design, but the *implementation* of an instrument: only a working prototype can provide the type of user testing and feedback, that could determine whether a particular affordance is relevant to a performer or not. In this scope, this translates to the question - to what degree should the implementation of designs, capable of digital audio sample reproduction, be addressed? The answer often lies not in what is possible, but in what is economical (also in terms of time): should one build logic gates from discrete transistors; or wire-wrap a CPU (central processing unit) from 7400-series logic ICs (integrated circuits); or write the program of the operating system (OS) for this CPU directly as machine opcodes? Clearly, some form of technology reuse naturally lends itself as a possibility in this area. On the other hand, in today's world enough capital can acquire, outsource and manage the design and production (or any other) stage, involved in bringing a finished digital technology to the market. This, however, also implies the disappearance of products and platforms from the mass market, as soon as they outlive their profitability; one way to address this would be to seek to understand such platforms in as generic terms as possible, and to reimplement them using available technology. Hence, the pursuit of implementing a digital audio workstation, necessarily forces the consideration

of an economical and generic implementation of a digital audio sampler as an issue.

In the past, this may have been a rather expensive sport – note that the first commercially available digital sampler, the Fairlight, at its introduction in 1979 [5] cost US$ 29,000 (adjusted for inflation, this sum is equivalent [6] to ≈91,000 US dollars in 2013!). In research departments, investments in new technology of that scope are both welcome and highly relevant; and while the voluptuous price tag may induce a degree of veneration among researchers and other users of the technology - for the same reason, it also hinders experimenting with modifications of a given device, colloquially often known as "hacking". After all, no-one wants to solder wires inside a machine costing x thousands of dollars, and thereby risk losing the device or its associated warranties - just to experimentally confirm whether a vague idea about an approach might work. However, the conditions – in terms of technology available for this kind of research – are somewhat different at present day.

On one hand, in the past decades we've seen a near fulfillment of the predictions of Moore's Law [7] about smaller, faster and cheaper digital electronics; this has made digital computer technology, in varying shapes (including smartphones), ubiquitous in the modern world - and as such, more accessible than in the past. On the other hand, the economics of cheap, but plentiful mass-market products, necessarily favors consolidation into big industrial players, and so fosters an atmosphere of protectivity, not the least in the domain of intellectual property. Academia is by no means excluded from this: for instance, A. Huang describes in his 2003 book "Hacking the Xbox: an introduction to reverse engineering" how his book research required the retrieval of a secret key from a Southbridge chip – which led to publishing problems with MIT (Massachusetts Institute of Technology), arising from the legal risk of contestation by the Microsoft corporation [8, p. 134]. Clearly, no researcher would ever want to implicate their employing institution in legal struggles, including those stemming from interpretation of intellectual property — trademark, copyright and patent — law. However, this protectivity may have also triggered a reaction in parts of the electronics and computing communities, in the form of work on, and public releases of, free/libre/open-source software (FLOSS) and hardware. The open-source community has also experienced growth in the recent decades, likely reinforced by the very same results of Moore's Law: increased public ubiquity of, and access to, computing devices - and the code that they run.

Thus, if a model for a generic digital sampler is sought, for the purpose of future integration into a digital audio workstation system - it would be beneficial if this model, at the same time, addressed connectivity to a personal computing device. This would allow possible DAW designs, based on such a model, to capitalize on the modern numerous omnipresence of computing devices. The most obvious model (intuitively speaking, but also for reasons outlined further in the thesis) for a sampler digital audio device, that works in concert with a personal computer (PC), is the *sound card* (alias *soundcard*). For economical and legal reasons, consulting an open-source design of a soundcard

would thus be well advised; a basic assumption at the start of this PhD project in 2007 was that the growth of open-source technologies would have generated such content already at that time.

This assumption was, however, wrong: searches at the time resulted with no obvious projects advertising open designs of soundcards, academic or otherwise. This is what reduced the original scope of this thesis, from the design of a DAW with emphasis on live performance, to the reimplementation of a soundcard as an open design - considering it as a necessary step, towards open designs of DAWs. Partially, this thesis represents an academic tutorial into soundcard implementation with different, relatively cheap, technologies: related papers in part II are all accompanied by associated webpages, containing released source code and media files (e.g. videos). These papers can be seen as series, discussing soundcard systems of progressively increasing performance quality (though not chronologically), represented through the metric of audio streaming parameters (sampling rate, sampling resolution, and number of channels) – as noted on the overview on table 1.1 (further on, a comparison of technologies is given on table 4.1).

Table 1.1: Overview of open soundcard projects in part II; stream quality is given as: sampling rate / resolution / number of channels

Project short title	Stream quality
Extending soundcard *paper II-A, 2010*	12.7 kHz / 8 bit / mono
Minivosc *paper II-B, 2012*	8 kHz / 8 bit / mono
AudioArduino *paper II-C, 2011*	44.1 kHz / 8 bit / mono
AudioArduino analog board *paper II-D, 2011*	(same as AudioArduino)
Audio bare-bones FPGA *paper II-E, 2011*	44.1 kHz / 8 bit / mono
Open s.card as lab platform *paper II-F, 2014*	N/A
Soundcard comparison *paper II-G, 2015*	44.1 kHz / 16 bit / stereo

While circuity for computer-based audio may go back as early as 1957 [9], it is important to recognize that the soundcard originally was embodied on the mass market as an add-on card for specific IBM PC architecture products, not necessarily as an implementation of an academic abstraction: the first soundcard with dedicated ADC (analog-to-digital conversion) and DAC (digital-to-analog conversion) circuitry for digital audio reproduction on PCs, the Sound Blaster by the Creative Labs company, was considered a common denominator in that business food chain already in the 1990s [10].

The effect of technological change should also be recognized - in the 1980s, it

may not have made sense to talk of a soundcard (in modern terms) for popular computer platforms of the time, like e.g. the Commodore 64: not so much because of the low (by today's standards) CPU clock speed, but because it already contained a sound chip, the MOS Technology 6581/8580 SID [11], which could have been used to demonstrate digital audio reproduction - albeit with low fidelity. Similarly, in the 1990s, it may not have made sense to talk of an open-source soundcard: not so much because of lack of existence of open-source software, but because it would have been difficult (i.e. *prohibitively expensive*) to obtain information on assembling a *working* open operating system, which would have allowed conveniences for development, matching those in proprietary OSs. The tutorial aspect of this thesis is thus a testament, that a synergy of technological changes had provided enough of a base, in the period 2007-2010, to allow for a relatively straightforward derivation of open soundcard designs: not only due to emergence of GNU/Linux OS distributions (e.g. Ubuntu, Red Hat) that, in spite of problems [12], reached a point where no bugs seriously impeded the basic user experience; but also because of emergence of open hardware platforms like the Arduino; and the growth of the Internet: here, most crucially because of developments in blogging and forum platforms, Q&A websites (e.g. Stack Overflow and related [13]), and public open-source repositories (e.g. SourceForge and GitHub [14, 15], providing content indexed by search engine giants (e.g. Google) – allowing for previously unprecedented access to information related to the usability of open source systems.

Academically, the implementation of a soundcard would require background from both electronics engineering and computer science. Education programs in these disciplines, while intimately related, usually can only afford to present their specific perspective, especially in undergraduate programs. The release of the open-source content in this project, can thus be seen as a base for development of a low-cost, cross-disciplinary, laboratory exercises related to these fields: provided the tools in the electronics lab bench, supplemented with a PC capable of running the noted OSs, are already invested in - exercises can be developed ranging from purely software ones, requiring no additional financial investment; to hardware exercises, where the most expensive parts (e.g. an Arduino board, or a Xilinx IC chip) are in the range of 30 to 50 US$ per unit. This price bracket would make the released content relevant to digital audio hobbyists and enthusiasts as well (regardless of how small a percentage of the general population they may represent). Such exercises don't necessarily have to maintain a digital audio perspective, even if that is their main intent - they can just as well be seen as illustrating general issues in e.g. real-time data streaming; this educational perspective is further elaborated in paper II-F.

The wisdom in choosing to implement an open soundcard can be questioned, especially in light of preexisting literature: considering e.g. [16, 17], white papers from Cirrus Logic Inc. (a fabless semiconductor company) and their references to Audio Engineering Society (AES) literature, is this not a case of reinventing the wheel? While such content does seem open enough, most associations of professional character have to operate under market conditions. As such, a vast

majority of such literature is not *open access* [18], but is instead behind a paywall in terms of Internet access; although this, arguably, is not a problem for well-funded research departments. More importantly, even if most of the literature *was* open access, inclusion of designs from business entities in open source content without explicit permission can be questionable: after all, no-one wants to see their intellectual property included in an open source project, which - while mostly intended to be offered *gratis* - can also, in principle, be *sold*. That is why the tutorial part of the thesis attempts to maximize the use of open content, where derivation from "first principles" doesn't apply – and otherwise, make a note where use of proprietary technology is inevitable. But ultimately, this open soundcard project had to source its parts on the free market just like any other; thus, by virtue of informing, the tutorial part necessarily also advertises output of companies like Intel, Atmel, Xilinx, Arduino etc. This could be considered a prime example of Internet's Poe's Law [19], but with an ironic twist, that could be formulated as: "*Even with a disclaimer, it is impossible to distinguish between mere information and advertisement, where products are concerned*".

An open soundcard tutorial, in its practical relation of topics from computer science (such as OS architecture and event timing) and electronics engineering (such as digital communication protocols, or pulse-width modulation (PWM) as de facto the cheapest method for analog audio output in a digital environment), can be more than a stepping stone to a DAW musical instrument implementation, or a base for an educational engineering laboratory exercise with a slightly lesser chance of vendor lock-in. In part III, a sample of papers from wider academic media technology research are included, where some of them might not concern themselves with audio as their main problem – but where all use the soundcard as a crucial part of the research tool chain. Availability of an open soundcard platform could have assisted each of these projects: for instance, in paper III-K, a high-end soundcard is used only to reproduce audio, while force-sensing resistor (FSR) sensor data is acquired using an Arduino with a low sampling rate. An open soundcard platform, where one could bypass the input coupling capacitors, would have allowed the sampling of the same sensors with much higher resolution, allowing for better quality research data; this bypass is an intervention on a hardware level, which could be hazardous applied to high-end (expensive) products - but, as paper II-A demonstrates, it is possible to implement relatively cheaply in a do-it-yourself (DIY) platform. Similarly, other projects reported in part III could have benefited from an open soundcard platform – and that would extend to all research projects of similar nature.

It should be admitted, that in spite of covering major issues in soundcard implementation, the tutorial part does not reach a "holy grail" of sorts, a self-imposed milestone, of demonstrating the operation of a high-fidelity – Compact Disc (CD) quality – soundcard. The reasons behind this, mostly (but not exclusively) technical, are discussed in paper II-G. That, however, is possibly not as important, as the note that – again, ironically – the same strong technological development process that provided the grounds for carrying out the work in this thesis, has *already* obsoleted some of the technology discussed in part II.

In fact, similar processes are observable elsewhere, not the least in pop music industry: the listening habits of the population, changed by technology, may have caused enough losses to the pop music industry, to force it to focus more on live events [20] - which may additionally validate the original emphasis on live performance in this research process. Considering that Moore's Law may be peaking soon [21], which might lead to a plateauing in all electronics-related industries (including musical instrument and media technology), this thesis also touches upon issues of obsolescence and diminishing returns in this area, among other possible future perspectives of an open digital audio platform.

1.1 Thesis outline

This thesis is organized as follows: in the introductory part I, the methodology is discussed in the next, section 1.2; the motivation is given in chapter 2. The DAW as a paradigm in live performance of looped electronic music is elaborated in chapter 3, through a discourse on established live interaction approaches with classic drum machines in section 3.1 (with details in Appendix), and with DJ setups in section 3.2 – resulting in a proposal for merging these approaches on a DAW platform in section 3.3, concluded upon in section 3.4. The contributions of the articles included in this thesis, focused on the open soundcard perspective, are argued in chapter 4: section 4.1 considers primarily the educational perspective of the work in part II, while section 4.2 considers the advantages of open soundcard technology use in research, exemplified by publications in part III; and diverse perspectives on open development of technology, which ultimately underlie and influence the development of open soundcards as well, are reflected upon in section 4.3. Finally, the conclusion along with future perspectives, is in chapter 5.

Note that as the papers in part II refer to each other, papers II-A to II-G appear as citations [1a] to [7a] in the respective bibliographic references.

1.2 Methodology

Chapter 2 is meant to historically outline the technology tree, that led to some of the proposal ideas in chapter 3. The autobiographical character of chapter 2, which outlines the motivation behind what become requirements for a DAW adapted for live performance, imparts an ethnographic methodology (cf. [22]) aspect to this thesis. But, as the response to a digital music instrument problem here is the development of open soundcard systems, which is technical in nature – in general, the major part of the Ph.D. work can be said to have followed an engineering methodology, as applied to both electronics and software development. Each of the papers in part II, as a project, is a standalone study of particular software or hardware aspects in soundcard development, which can be seen as a technical experiment; the papers in part III are likewise

standalone studies, but each with methodologies respective to their field. The projects typically passed through stages of design, implementation, verification, and analysis of operation through data collection; while the design stage in electronics (where appropriate) is more distinct – in software development, it was often conflated with the implementation stage in iterative, rapid development cycles; close in spirit to some aspects of established software development methodologies such as AGILE [23].

The verification and data collection and analysis stages involved quantitative research methods. In part II, besides the basic literature overview for each project, the *data collection* process in terms of hardware involves obtaining measurements through instruments like multimeters, but more importantly, digital oscilloscopes - which allow download to a PC and further computer-based analysis; in software terms it involves capturing programs' output, often timestamped, in log files. The *data analysis* phase in both hardware and software mostly revolved around time measurements, such as determining temporal difference between two events, or finding signal period (i.e. frequency). However, for software analysis in particular, the research process at times required custom visualization, which then yielded development of customized software tools; this is most apparent in paper II-G. Beyond this technical research core, papers in part III additionally demonstrate a methodology common in the field of human-computer interaction (HCI): usability evaluation through user testing, and statistical analysis of the results thus obtained. Had the project resulted with prototypes of relevant quality and reliability, usability testing would have also been a critical milestone in the development of both an open soundcard - and a DAW for live performance, based partially on its technology.

Chapter 2

Motivation: a labor of angst

> *"... writing about music is, as Martin Mull put it, like dancing about architecture."* [24]

If this is the section where the thesis author is allowed a degree of autobiographical reflection, I will take the opportunity and switch to a first-person narrative. As my Curriculum Vitae shows, I have had formal connection with both music and technology, in varying degrees throughout time. In fact, the main motivation comes from my younger time, when I was intimately involved with music and recording studio scenes and communities: above and beyond assisting musicians in general with an improvement of an instrument, the DAW for live performance - as described further on in this thesis - was something that *I personally* would have wanted to use, for a potential musician's career of mine. In my younger days, I would have certainly described the efforts towards this instrument as a "labor of love"; but with age, comes cynicism - and if I'm to be honest today: to a comparable degree (if not higher), these efforts could just as well be described as a labor of *angst*. Angst notwithstanding, the motivation is also a product of all the good, positive and exciting experiences I've had at the time - including the influences of music artists and producers that I have listened to and appreciated. This section will describe some of those influences, that have had the strongest impact on the development of my particular idea of the DAW as a live instrument.

2.1 Baby steps

Technology-wise, I have since earliest childhood had interest for what I would describe as media technology - technology that, unlike most inanimate matter, would directly appeal to my perception: technology that could record and reproduce speech, music or images (moving or not). In other words, while I could clearly notice the utility in devices as an electric refrigerator or stove, such

examples of technology never piqued my interest, as much as a black & white TV set, a mono radio/tape-recorder, or a diaslide projector did. Maybe more accurately, what fascinated me about these devices are two factors: that they could convey a message meaningful to me, and not only that - they often have allowed me immersion in that message's implied context (in a more natural way than, say, printed media did). For instance, one of the most immersive experiences for me during childhood, that I can still recall, is listening to the Muppet Show Album (1977) vinyl record on a mono gramophone; the music just sounded so "crazy" (yet amicable) and different from my daily experience - it could grab my attention to the point of immersion on its own (rather than by association with the Muppet Show characters, which I also liked). This in itself was a source of wonder to me, as immersion was the unexpected end result of combining a black vinyl plastic disc, with a suitcase featuring a metal spinning disc and a plastic arm: both, on their own, rather mechanistically inanimate objects.

Thus, I (like, presumably, many others) came under the spell, of what is maybe best expressed through the wording of author Arthur C. Clarke's Third Law: "Any sufficiently advanced technology is indistinguishable from magic" [25]; in this case, the "magic" of inanimate objects producing messages that can affect people personally. And the infatuation with media devices hardly stopped at the gramophone. I experienced the arrival of two new products in the early 1980s which left a profound impression on me; both of which carried some prestige to owners at their introduction.

The first was the VHS (Video Home System) video-tape recorder: I found it absolutely mind-blowing that this machine could both record and reproduce television programs, and you could have it at home. Disappointed at my parents' refusal to acquire one for many years, I spent quite some time pondering (having realized that both audio and video cassettes alike contain magnetic tape) how I could possibly modify a mono audio tape recorder to record TV video signals – so as to avoid spending money on a new VHS recorder. This thought experiment certainly contributed to my choice of electronics engineering studies later on. The second profound product for me was the Commodore 64 computer: not only it controlled both the audio and video signal of a TV set, it could use that to reproduce diverse games, loadable from an audio cassette tape – and more than that, it offered the user a possibility to eventually program and implement their *own* games (or more generally in today's terms, multimedia productions). I was rather excited with these possibilities, and after my parents got one from me (not the least due to permanent nagging from myself), I spent a lot of time with the C64. Notably, most of this time was spent in a state, that the current Western legal outlook would consider *piracy*: while the Commodore company may have had representation/distributor in my country of origin at the time, the game software producers definitely didn't; and the only way to get new software was to copy it from someone else: from their tape to mine (given that blank audio cassettes were readily available at the time). However, I also spent

a portion of the time buying magazines, typing in the source code they printed of diverse small programs and games, and running it; this would presumably be seen as a legal customer action even today. This computer allowed me early contact with BASIC and assembler programming languages; though I must admit that I've forgotten most of these Commodore experiences in the decades that followed.

Music-wise, my beginnings as a child are in the mid portion of the 1980s, when I was signed up to elementary classical music education, which involved both voice training and piano lessons. While I was mildly impressed at myself, once I could play the piano with both hands without effort – the kind of classical music used for training didn't appeal much to me, and moments of immersion were rare. Slowly, I started feeling it as a burden, and ultimately asked to be relieved of the music school attendance duties. A short period of no particular interest in music followed, until I hit puberty, around 7th grade elementary school (which would be the period before and around 1990). This approximately coincided with the introduction of so-called "satellite TV" in the local area where I used to live: essentially, inhabitants of residential buildings would pool money together, and buy a satellite dish and receiver equipment, whose signal via cable was then distributed to households. This allowed, in addition to the state channels, up to 4 additional foreign TV channels on a household TV; most notably, the Music Television (MTV) channel was typically offered in such a package, and that is how I got in touch with it and its offerings. In this period, I discovered the hard rock band Guns N' Roses, and their hits from the "Appetite for Destruction" (1987) album – and I was instantly hooked on the heavy sound of the distorted guitar.

2.2 Pedal to the metal

I hardly stopped with Guns N' Roses, who merely spawned an interest into ever so harder expressions of music; and soon after, I discovered the heavy metal band Metallica with their "...And Justice for All" (1988) album; which, along with their previous releases, officially turned me into a "metalhead". Expanding first to bands like Motörhead or Iron Maiden, my interest was then led to what was known as speed or thrash metal genre, especially through albums like Bathory's "Blood Fire Death" (1988), Kreator's "Extreme Aggression" (1989), or Sepultura's "Beneath the Remains" (1989). Ultimately, I settled on primarily honing my tastes for death metal, grindcore and noise; consuming releases such as "From Enslavement to Obliteration" (1988) by Napalm Death, "Realm of Chaos" (1989) by Bolt Thrower, or "Left Hand Path" (1990) by Entombed (and occasionally, slower, doom metal variants like "Forest of Equilibrium" (1991) by Cathedral). Titles like these can certainly be read as a form of teenage rebellion, or even angst - although, I feel for me it was more a way to deal with feelings of increasing inadequacy in respect to wider society, through reinforcing an escapist sense of *weltschmerz* in a (somewhat) small community of like-minded

people, distinct from what would have been locally recognized as alternative music at the time (for an analysis of the local musical output in the generation preceding this one, consider [26]).

Having so deeply ingrained myself with a music scene of this character, which very clearly revolved around the sound of distorted electric guitar, I was naturally motivated to take up playing the instrument; this time, however, because of sheer desire to be able to play. Musically, one of the things that I found so exciting in the electric guitar, was not so much the capability for melody and harmony, inherited from its acoustic ancestor; but the capability to function as a nearly *rhythmical* instrument, especially with the metal playing technique of palm-muting; which comes most prominently to life, when involving power (fifth) chords - and, especially, distortion. I found that this enforces a specific relation between the performer and the instrument: the instrument is no longer just the guitar, even if it is electric: the complete instrument is composed of the guitar, a distortion pedal, and an amplifier (plus assorted cables, and of course, access to the electrical power grid). And besides the conceptual shift in considering this system as an electronic instrument (even if the guitar is still played, in many respects, in the traditional acoustic manner), such a relationship also means that this instrument becomes somewhat more expensive to own and play, especially for a teenager.

2.3 Ridin' on the wings of inflation

While mentions of expenses in this context rarely deserve a response, other than "*Get a job!*" – this period, unfortunately, coincided with what was known in South-Eastern Europe as "transition". My first memory of this is inflation, maybe most vividly captured on Fig. 2.1. The size of the banknote denomination on Fig. 2.1 is maybe not so impressive, compared to contemporary currencies of functioning economies like the Japanese yen; however, it is a symbol of the times when money was worth less by the day, and the desperate response of the government to add more zeroes merely reinforced the perceived loss of value; at the end of 1989, the inflation rate was [27] around 1300 %.

Fig. 2.1: Left: A banknote for $1{\cdot}10^5$ dinars, 1989 issue by the People's Bank of Yugoslavia. The same year, the largest banknote issued was for $2{\cdot}10^6$ dinars; the previous year, the highest denomination issued was for $5{\cdot}10^4$ dinars. Right: The next year, 1990, there was a revaluation of the currency, resulting with the same amount being represented as 10 dinars.

With the independence of Macedonia in 1991 and introduction of own currency in 1992, the inflation may have stabilized a bit; and even if the country was spared from the brunt of the chaos in the rest of ex-Yugoslavia, things hardly improved drastically. This is not to say that I was living in poverty, which I wasn't (after all, my family could afford to send me for a year to the USA in 1994) – however, those were otherwise the real prospects for many, if not most, people there at the time. With that in mind, most people at the time (including my family) simply couldn't afford to invest in anything beyond assurance of survival; and musical instruments, just like entertainment technology, was considered a luxury. And that I got a taste for the electric guitar in that period, clearly wasn't helping anyone.

Of course, I would have preferred to invest my own money, bypassing the approval of my family; and so I did get the idea of getting a job already then – especially inspired by the American notion of teenagers "flipping burgers" to earn for instruments. However, consider the general economic depression of the times: for instance, the CIA World Factbook for 1995 [28] states an inflation rate of 54 %, unemployment rate of 30 %, and a national GDP per capita of 900 US$ (which amounts to some 75 US$ per month) in Macedonia; and it wasn't better in the early 1990s. What I saw, was the same other young people did: that there were no jobs available - and even if there were, one could hardly save even the entirety of such salaries towards equipment costing hundreds (if not thousands) of dollars, in a time frame less than years. So much for "getting a job"; which was especially irritating after I visited the USA, and saw for myself that "flipping burgers" was indeed a real option to teenagers there at the time. Interestingly, the idea of DIY as applied to electric guitars didn't quite appeal to me at the time, even if I was aware of the work of luthiers in the country, such as Branko Radulović. I think it is because I saw it as mostly involving woodworking, which in itself requires quite a bit of knowledge and equipment; but in which I didn't have as much interest as in electronics, where I had already started to get invested in.

2.4 While my guitar gently weeps

Still, through a combination of savings, begging and perusal of the second-hand market, I managed to acquire a rather worn-out electric guitar in the early 1990s (Fig. 2.2) - which I, in a fit of egotism, lovingly named "the Smikicaster" (a portmanteau of my nickname at the time and the names of the famous models of the Fender company, which ended up as a nice little in-joke).

With the acquisition of an electric guitar, I could participate in bands in the local scene. I should note that given the underground scene at the time was rather small, it often times allowed for a degree of genre cross-pollination; and thus, aside from a brief appreciation of the grunge scene, as exemplified by Nirvana's "Nevermind" (1991), I also got exposed to punk/hardcore releases such as D.R.I.'s "Dirty Rotten EP" (1983), Youth of Today's "Can't Close My

Fig. 2.2: left: concert of the hardcore band Dead Cops Rock, Codex club, Skopje, 1992 - on stage, from left to right: me, Zoki (vocals), Slave (drums), Iko (vocals), Meto (bass); right: me, "rocking the place" with my guitar, "the Smikicaster" (VHS captures, [29])

Eyes" (1985), Bad Religion's "Suffer" (1988) or Kromozom 4's "Rien Ne Sert De Gâcher De La Bande, Il Faut Faire Cuire Son Steak A Point" (1987), while still retaining my tastes for metal. One of the attractions of this kind of music, for me, was definitely the fact that I didn't need to be a master guitar instrumentalist, in order to sound acceptable (even according to my own criteria) for that particular aesthetics. In fact, instead of metal, at the time in Skopje I mostly ended up playing in bands such as the hardcore "Dead Cops Rock", or the experimental projects with Filip & Dimitar Mitrov "V.I.S. Šegobijci", "Arpadžik" - and "Nekrojagotki", with its demo album "Majčice veštice". I presume most of the material (like audio cassette demos) that I was involved with from this period has long disappeared; Fig. 2.2 is a memento of those times.

The crossing between genres may have been facilitated by two factors: for one, the underground music scene was small, so "everyone knew each other" - and it was thus difficult to sustain clashes between genres, akin to the ones in its native Western environment (as in e.g. "punks" vs. "hippies"; here, enmity was mostly reserved for external *poseurs*); and secondly, this scene (just like for C64 software mentioned earlier) operated mainly by copying audio tapes - that is, by piracy. However, the analog method of copying audio signal from/to cassette tapes, meant that each successive generation of a copy accumulated more noise than its originator (see e.g. [30]); resulting with practically unusable copies several copy generations "downstream" from the original recording. Thus, owning an original record in those days was seen as somewhat of a status symbol, and a small second-hand market for such imports did emerge. And from that – for me, as well as for nearly all of the people I've played music with – it didn't take long, for the ambition to record our own music to develop. But while owning an electric guitar may be enough for band rehearsals and concerts, it most surely isn't enough to perform recording. The cheapest and most straightforward way to record a band demo at the time was to utilize a cassette tape recorder with a built-in microphone; which, given that there

was no way to adjust the volume of individual instruments on the recording, typically resulted with a rather poor quality live recording. And from that point on, my interest in technology was strongly focused on music processing and recording equipment (even if, incidentally, poor quality recordings were not necessarily shunned by subcultures with a DIY aesthetic, like the hardcore scene).

2.5 Digital audio arrives

By the mid 1990s, however, the emergence of digital audio products seriously disrupted the understanding of copying inherent in analog methods. To begin with, the introduction of cheap CD players popularized the notion of the CD as an undisputed source for first-generation copying to analog tape: especially because the signal quality does not deteriorate by mere use of the medium (which is otherwise inherent in the audio cassette), and digital signal correction ensures that small scratches on the surface do not have the same effect on audio reproduction as in the case of the vinyl phonograph record. Then, the emergence of Creative Labs' soundcards, in particular Sound Blaster 16 in 1992 and later generations, allowed for a popular shift in perception of the personal computer as an audio device. Up to this time, the idea of the PC in the studio was mostly established as a Musical Instrument Digital Interface (MIDI) controller, driving sound reproduction in external synthesizers; with the Atari brand being quite popular in this context. With the introduction of the soundcards, the IBM PC architecture (which started becoming increasingly more affordable) began to be seen as platform that allowed manipulation and reproduction of digital audio with CD quality, with unprecedented precision (down to a single sample) and without the deleterious effects of noise.

I may have penanced slightly for my lifelong participation in music piracy, by using the opportunity of my visit to the USA as an exchange high-school student in 1994/95, to spend nearly all my allowance on acquiring original CDs. In the USA, I also acquired a small 4-track analog recorder, which could use the space for two sides of audio stereo signals on a normal cassette tape to store 4 mono signals, and had a mixer interface with a channel strip for each mono channel (I cannot recall the brand, unfortunately). This recorder did find a limited use after I got back to Skopje in 1995, but already then it was clear that this device was more a remnant of the past, than a tool that would allow for serious music production. Eventually, the emergence of relatively cheap CD writers and writable CDs allowed music fans identical copying of audio on a digital level, without any signal degradation with noise; which finally pushed the audio cassette toward obsolescence.

2.6 Genre expansion - folklore and electronic music

For me, this technological change was also followed with further changes in musical interests; and the expansion towards different genres for me didn't stop at styles that involved the heavy, distorted guitar. In fact, while involvement with rock'n'roll-derived forms of music can be seen as an expression of values that could be seen as modern and international (that is, values identical to particular Western subcultures), there was also a motivation to look inwards, towards music that would be independent of that heritage, and thus unique in a global scope. Thus, I ended up studying with a group of people as students of the *doyen* of Macedonian folklore music, Pece Atanasovski, until his passing in 1996; subsequently, the student group evolved into an orchestra, that carried the late teacher's name (for related information, consider [31]). Given my guitar background, I took up the study of the string folk instrument *tambura* in that group (see Fig. 2.3). While the general outlook towards history and tradition wouldn't necessarily relate to music technology, playing in a folk orchestra exposed me to the concept of ethnographic *field recording* (and the value of portable audio recorders in that context) – and subsequently, allowed me the unique experience of participating in recording an acoustic album (which was nonetheless recorded and mastered digitally) in a "real" studio. Additionally, I was exposed to a culture, where songs may have existed for centuries; but due to the nature of oral transmission of folklore, authorship copyright as understood today is not applicable - simply because the actual author is not known.

Fig. 2.3: An incarnation of the Orkestar Pece Atanasovski during the release of the self titled album (remarkably enough, it included members of the *other* hip-hop band in town at the time, Čista Okolina). From left to right: Risto Solunčev (gajda), Gjorgji Donev, Vančo Damjanski, Viktor Siljanovski (kaval), Dejan Spasović (kemane), me, Vladimir Martinovski, Dejan Sibinovski (tambura) and Vele Solunčev (tapan). From the CD booklet.

However, I had also been exposed to genres that explored expression beyond the allowances of electric guitar, or indeed, any heritage from acoustic music instruments; and could be considered fundamentally electronic music genres. To begin with, I was exposed to what was variously referred to industrial or EBM genres, which some fans from the metal and hardcore scenes listened to as

a side indulgence (and as such, I cannot recall a local scene exclusively dedicated to these genres at the time). Therewith, I gained appreciation of albums like "Psalm 69: The Way to Succeed and the Way to Suck Eggs" (1992) by Ministry, "T.V. Sky" (1992) by Young Gods, "Kapital" (1992) by Laibach, or "05:22:09:12 Off" (1993) by Front 242. Initially, I appreciated this kind of material solely in terms of a studio recording intended for reproduction (playback); with my background pretense of a guitar instrumentalist, I simply found the concept of live performance of electronic music *laughable* (especially since I had witnessed events, where "performers" would shamelessly play back digitally prerecorded material, and otherwise merely *pretend* to turn buttons and perform live - in the times when audiences wouldn't notice, as they'd mostly recognize attempts at playback by the presence of audio cassette noise; for similar sentiments, see [22]). On the other hand, this kind of material exposed me to thinking about sound which was musical, yet beyond the usual musical essence: we could consider a musical composition traditionally as a collection of notes, each with a pitch, volume and duration, arranged in time; and put in music technology terms, a MIDI recording of a song will still convey its musical essence, even if the (pitched) instruments that reproduce those notes are interchanged, e.g. violin for piano.But in these genres, you might have a memorable sonic passage, which might not even have a recognizable pitch at all - but rather, a distinct evolution in time of the sound's spectral content, which in terms of instruments would be called *timbre*; except here with an extended meaning, so it would also encompass the difference between sound recordings, say, of a car and of a refrigerator. For instance, the more exciting moments for me were when bands would utilize a "montage" of samples of distorted electric guitars, however taken from entirely different albums, and thus with completely different productions of their sound spectrum; this would result with a sudden and rhythmical change in "timbre", which was very difficult (if not impossible) to reproduce as an electric guitar player (the closest would be to suddenly turn on a distortion pedal with your foot, and thus switch the timbre of the electric guitar from "non-distorted" to "distorted"; but rhythmical switching between multiple distortions, even if supported by multiple pedals, would require extremely demanding footwork while also standing and playing the guitar with your hands). Still, this influence was partially responsible for developing my interest in rhythm machines and samplers, as well as contacts with people focused more on electronic music instruments.

2.7 A hip to the hop, and you just don't stop

Another significant influence on me was the one of rap music and hip-hop culture. I must admit that initially, I didn't show much appreciation for this kind of music (for instance, when MC Hammer's "U Can't Touch This" (1990) became a hit, I was absolutely horrified that I had to hear that song everywhere; I first found appreciation for it years later). That was all to change with the

discovery of the band Das EFX, initially through the exposure of the videos from their "Dead Serious" (1992) album on the "Yo! MTV Raps" TV show. That brought the realization that rap can sound hard and heavy, while not referring to the dictionary of the hard and heavy music stemming from rock'n'roll I knew up to then (clearly, also here I was first attracted by angst) - and, on top of that, could even sound *funky*. This brought me in contact with the local (at the time, extremely tiny) hip-hop scene; and by 1993, I became a part, along with Goce Trpkov and Mitko Gaštarovski, of the rap band S.A.F. (also SAF, Fig. 2.4), where I had the role of a rapper.

Fig. 2.4: Boyz n the Hood: old S.A.F. promo image; from top to bottom, clockwise: Mitko a.k.a. Pikisipi, me, and Goce.

In this regard, I was mostly influenced by releases of what was known as the East Coast "new school", such as "Whut? Thee Album" (1992) by Redman, "No Pressure" (1993) by Erick Sermon, or "The Most Beautifullest Thing in This World" (1994) by Keith Murray. What impressed me here was that it was possible to rediscover new musical quality from old recordings: a short looped sample might serve as a standalone song background, where it could offer a significantly different aesthetics from the sample's source recording. And while the minimalist approach to composition in hip-hop could be perceived as "boring" in the classical music sense, it also heightened the sense of polyrhythmics that the rapper's diction formed with the beat, during the performance of the lyrics - which for me, at the time, was a newly found musical appreciation. But that wasn't all: rap music is intimately connected to the concept of DJ-ing, where two turntables and a mixer are used as a musical interface (for more, see [32]);

while I initially appreciated DJ scratching more as a form of acrobatics, what changed my mind most strongly was the work of DJ Premier in the group Gang Starr, in releases like "Hard to Earn" (1994) – whose sophisticated scratching in songs often left a musical impression of the same prominence, that guitar solos have in rock-based genres. Realizing the musical potential of the DJ equipment, I eventually got the opportunity to try it - and finally started fully appreciating it as a musical instrument: namely, it was hard for a novice to sound good playing live, as it was easy to make *mistakes*. The experience of recording S.A.F.'s debut album, "Safizam", brought me in contact with - among other technologies - both (some of the) "classic" sequencers discussed later in chapter 3, and digital multitrack recorders like the Roland VS880 (incidentally, this album gained somewhat of a cult status as time went on, maybe most obviously apparent at the concert for the 15th anniversary of its release, on 24th April 2015 in Skopje, which attracted nearly 5000 people in attendance).

While there were DJ techniques that were relatively easy to learn on turntables and a mixer (such as beat-juggling, where a loop is played out of two identical records, by rewinding the one record while the other is playing), it was clear to me that more expressive playing would involve a steep learning curve. Furthermore, the sensibility of the turntable styluses means that external factors can be just as influential in live playing, as the mastery of the instrument is (especially important in open-air performances, where e.g. wind, or feedback vibration from the live sound reproduction system on the floor of the stage, could cause a stylus to skip to a random location on the record). This is often addressed by adding heavy bases to the turntables (for instance, the legendary DJ turntable Technics SL-1200 weighs 11 kg); making the whole instrument (of two turntables and a mixer) less portable than, say, a guitar. But above all, with a starting price tag of at least 1000 US$ for the instrument (as turntables and mixer devices), which then merely serves as a gateway to a nearly endless need to purchase new vinyl records – say for up to 15 US$ a piece (and additional postage fees and import taxes on top) – I finally got discouraged in pursuing proficiency as a DJ instrumentalist. After all, I wasn't *that* interested in reaching the full scope of techniques a so-called battle DJ might express; I was mostly interested in relatively simple moves that I found both musically interesting and *intuitive* – which mostly revolved around the use of the mixer's knobs and faders in rhythm with the playing track(s) (such as the use of crossfader to "cut" between two different songs playing with synchronized tempo; which also results with a sudden change of "timbre" - or rather, sound spectrum - of the overall sound).

2.8 Design ideas emerge: electronic music instrument sessions

In parallel with the strengthening of my interest in hip-hop DJ-ing techniques, in the second half of the 1990s many people I knew, who owned bits and pieces

of audio equipment, engaged in borrowing equipment to each other for amateur, often impromptu recording sessions. I came to attend, and often participate in, many such sessions - even if the devices I could have contributed with (e.g., the 4-track recorder) weren't very interesting in that context. These sessions were essentially an alternative night out with friends - except not in town, but in someone's home studio; where diverse amount of synthesizers, drum machines, and possibly computers, would be connected to a mixer, and the final mix routed both to amplifiers and a recording device. Every participant would either program a sequencer track, or modify the synthesizer sounds or the mix, and that may have gone on for hours until all agree something sounds good; in which case a snippet of that production would have been recorded. It was in this environment where I met rhythm machines like Yamaha RY30 (first introduced 1991) and RY20 (1994), synthesizers like Roland Jupiter-6 (1983) and Juno-106 (1984), or Korg MS-10 (1978) and MS-20 (1978), and samplers like E-mu Emax (1986), SP-1200 (1987) and ESI-32 (1994), or Akai MPC2000 (1997). Here I firstly learned to appreciate that even if a synthesizer is driven digitally (via MIDI), if it has an analog engine it would still reproduce the "warmth" and fullness of sound expected of such instruments. Second, I started to appreciate the step sequencer interface on drum machines like the Roland TR-808 (1980): having 16 buttons to represent 16 steps in a sequence, it was relatively easy to insert (or remove) a drum sound at a precise location "live" (i.e., while the drum sequence loop is playing) by toggling the respective button; in contrast, on the Yamaha RY20, choosing the particular step involved multiple button presses, which in some instances may have required the beat to stop – which is why the only reliable "live" thing (in respect to sequencing) on this machine was to switch between pre-programmed drum sequences (which could only occur at a full measure). Samplers like the ESI-32 were rack-mounted units that were solely intended as MIDI-driven sound generators; and as such couldn't have been thought of as standalone instruments. All the good times notwithstanding, these sessions could end in frustration as well: often times it was enormously difficult to combine up such diverse instruments in a sensible way for a single performance context (at least, before the meeting would end) – and this was one of the main motivators for me, to start thinking about an extensible platform that would integrate features of drum machines (step sequencer, pad bank), samplers and analog synthesizer engines in a single machine with a mixer interface. I simply wished for something that I could power up, and after several minutes of booting, have the possibility to play with these features - *while* I still have the inspiration (in contrast, if you have an inspiration for a guitar riff and you have a guitar laying around, all you have to do is pick it up and start playing).

Communicating within this environment eventually brought me in contact with people from the, then nascent, EDM scene - in particular what was known as the techno & house scene. While I was previously exposed to releases like "Experience" (1992) by The Prodigy, "Musik" (1994) by Plastikman or "Selected Ambient Works Volume II" (1994) by Aphex Twin, simply by having been

informed about popular music – here I came to appreciate records like "Jill's Meth" (1996) by DJ Slip, "Kat Moda" (1997) by Jeff Mills, or releases by the Tresor (Berlin) label (again, attracted by angst). In fact, I cannot even recall what other artists I may have liked best; in this scene, the label may have meant just as much (if not more) as the artist - and after all, most fans were more concerned with consuming hours-long mixes by a trusted DJ, rather than individual tracks; accurate knowledge of artists was mostly the concern of DJs. And having no ambition of being one, I found that I was quite content with just asking to copy whatever I fancied, from what I'd end up hearing in that community - without putting too much effort into encyclopedic facts. However, I still continued participating in sessions, which eventually brought me to an event, which strongly reinforced the idea for an integrated hardware platform for live performance of electronic music.

Fig. 2.5: Advertising flyer for the live event in club Sachmo, 2000

Namely, the thought emerged that these sessions should be presented live to an audience, and so I, along with Aleksandar Ordev, Filip Mitrov and Samoil Radinski, was a part of a concert in the club Sachmo in Skopje, on March 8th, 2000. A flyer for the event is shown on Fig. 2.5, and there were about 50 (or less) people in attendance, for whom we prepared some techno oriented tracks. The concept was to prepare some drum tracks and MIDI sequences for the synths beforehand; the output of individual synths and drum machines was then taken to a mixer; and the mixer real estate was essentially split in two: one drum machine and a pair of synths (i.e. 3-4 channels on the mixer) would play one song, while another drum machine and synths would play the other.

The live playing then consisted mostly of real-time changes of either synth parameters, or of rhythmical track volume changes through mixer faders. Of course, using faders to rhythmically modify recorded music is not without precedent: for instance, consider the interventions of producer Lee "Scratch" Perry on the tune "Revelation Dub" (1976) by The Upsetters, which result with rather quick rhythmical fades of the vocals, in a track otherwise performed on traditional instruments.

The trick thereafter was to prepare songs that were close in tempo (or prepare suitable transitions), try to start the drum machines manually in sync, and then try to make a crossfade transition between the two songs like a DJ would do; except here, as there was no DJ mixer (and thus no main crossfader), we'd have had to do that manually on the main mixer (that is, 3-4 channels of the one song would be put down, whilst the 3-4 channels of the other song are put up). This may have been a necessity, because I think at least one of the drum machines was not able to load a new song (as a collection of drum sequence patterns) without stopping the currently playing one. This was a one-of-a-kind experience for me (I haven't had a similar one since), and the key findings from it were:

- It took as more than 2 hours to set up the instruments on stage, about 40 minutes for the performance, and more than 2 hours to disassemble the equipment
- Having two people perform a song crossfade (where each controls their own respective 3-4 channels) on a mixer makes sense musically; but bumping into each other on a crowded stage every 5 minutes isn't fun (as we'd go back to whatever instrument we were tending as soon as the crossfade was over)
- The only times we'd elicit a response from the audience (i.e. mild dancing and cheering) was during the song crossfades (about 40 seconds), plus about 40 seconds into the start of the new song; as we had to stretch the songs to up to 5 minutes, the audience quickly grew bored (and tweaking synth filters wasn't always successful in gaining their attention back)

While it is a form of success to elicit any sort of positive response *at all* (even if mild), this event certainly wasn't an extraordinarily positive memory for the audience. It is true that this outcome can be attributed to beginner's growing pains, and that further practice may have eventually alleviated those problems. However, I couldn't resist the thought that if we *each* had a single platform with a mixer interface, many of these technical problems would not have even been encountered. As a starting point, I imagined that a device like that should be capable of being "split internally": I should be able to program a (say) 2-track drum sequence and 2-track synth sequence, with individual outputs on channels 1-4, and have it considered as one song, driven by one tempo; with channels 5-8 representing another song with an independent tempo - and I

should be able to seamlessly mix from one song to the other, loading a new song in the muted section while the non-muted one is playing, continuously. Clearly, this "internal split" could extend to arbitrary number of channel groupings and tempos involved; however, the presence of a master crossfader would be most clearly defined in the case of a 2-way split. In fact, if the device would support a network interface (which, to an extent, implies an addition of an embedded computer), then a network protocol between such platforms could be devised, which would allow for at least two modes of group playing:

- A common tempo information signal is distributed to all devices, used to drive individual sequencers; the mixed output of all individual devices is collected and mixed in equal measure
- "Token Chain": Player A has the "token", so the tempo of A is available to all others, while A is playing own sequence, with own crossfader fully to the "left". The system is informed (either pre-programmed, or on the spot), about which player has the token next, say player C. As A starts moving the crossfader from "left" to "right", the sequence player C is playing (who keeps own crossfader to the "left") starts being mixed in. As soon as A has moved the crossfader fully to the "right", the "token" passes to player C, which means player C is now fully in control; which will last until a new next player is chosen, and C moves own crossfader to the "right" - to pass the "token" to the next player in the "chain".

The "token chain" approach would essentially allow for a DJ-like crossfading experience between songs, except where songs are performed live by individual artists; with passing of control, similar to the approach for managing performance of multiple DJs known as "back-to-back". However, it also increases the problem domain of such a platform development: in addition to analog and digital signal processing, one would have to consider embedded computing and networking as well. The machine was otherwise imagined primarily as a multichannel audio sampler, which would be driven by the independent sequencers, with the resulting sound assignable arbitrarily to individual mixer channels faders. While such a device would principally drive external synthesizers via protocols like MIDI – implementing it as a customizable platform would have allowed insertion of analog oscillators and envelopes, allowing for synchronized control of both digital and analog synthesis from a single unit. Around this time, I saw the introduction of Roland VS880 (1996) Digital Studio Workstation (multitrack recorder with fader interface), and Yamaha 01V (1998) Digital Mixing console, both of which controlled multiple channels of digital audio, through motorized faders that could be driven via MIDI. As both of these were sometimes colloquially called "workstations", I thought this as an appropriate category for my imagined platform, even if it was meant to address issues in live performance (unlike the main functions of the VS880 and 01V). Still, motorized faders are a powerful visual effect, and they could have well been used in the platform, especially to indicate cross-fading: by moving the

crossfader left-to-right, the group of individual channel faders for the one song would automatically move down, while those of the other song would move up.

Sadly, this was to remain a mere vision, even if since 1995 I have been enrolled in bachelor university studies of electronics & telecommunication at the Electro-Technical Faculty in Skopje. The program there used to have heavy focus on theoretical math and physics, which I found rather tedious and straining at the time; by now, however, I have learned to appreciate that approach – especially since that very background, is what allowed me to pursue the research in this thesis in the first place. The labs, however, I found generally unappealing; mostly because none of them (that I can remember) dealt with practical reproduction of either sound or video - at least not in a context that I could personally take further, to independently develop a DAW platform. While I didn't stop hoping I would eventually gain the practical knowledge of building such a platform up to my graduation in 2001 - in retrospect, the problem is that such development requires practical, "catalog" knowledge of parts, as well as means of sourcing those parts; and it was difficult to find related information in a systematic, tutorial format on the (then, expensive and dial-up) Internet (as the only alternative to the university, which did not concern itself with such issues).

2.9 Further developments and opportunities

While I ultimately had to abandon the idea for actual development of the DAW platform, there were a few additions to the idea during the late 1990s and the early 2000s. In that period, I, together with Samoil Radinski, Ognen Uzunovski and Ivan Todorovski, was a part of a promotional agency for techno music events called Balance (see Fig. 2.6); there I mostly had the role of a multimedia (web, graphic and audio/video advertisements) designer.

Fig. 2.6: Together with Balance promotional agency, somewhere in year 2000; left: me, Samoil, Ognen, and Ivan; right: me, Derrick May, Ognen and Samoil (enumerated from left to right on both images)

The first booking the agency made was for the international star DJ from Detroit, Derrick May; whose performance in 2000 profoundly changed my view of techno DJ mixing. Namely, while at the time I had gained great admiration for hip-hop DJs, most of the techno DJs I found bland in terms of handwork, even if I liked their selection: they would mostly perform slow crossfades, and the rare scratching was mostly simple, and even more rarely sounded appealing (especially with fast tunes, with tempi >120 BPM). Derrick May, on the other hand, used quick, rhythmical crossfades, which in the most successful moments could even sound as a meta-composition over two playing songs; supplemented with motions like stopping a record by slowing it down in the last beat of a measure (which could be compared to a scratching motion), before a quick transition into the same (or different) song. This kind of performance I could very well appreciate; at least in terms of fader operation, the envisioned DAW would have been able to support similar kind of operation. But, an addition of a turntable interface, whose rotational speed controls both the sequencers' tempi and the speed of sample reproduction, to the platform, would have also allowed for a DJ-scratch-like control of sound from *live-playing sequences*; and could thus integrate more closely with the production workflow of hip-hop, techno and many other related DJ cultures.

In fact, it didn't take long for aspects of this idea to be validated by the music instrument industry: already in 2001, the Final Scratch platform [33] started emerging, which allowed for a system where turntables playing special time-code encoded vinyl records controlled playback of digital audio files from a computer. The close contact to the DJ cultures also motivated me to wish for a solo career in a similar sense: while I wasn't exactly interested in being an actual DJ who mixes other people's records – I was inspired by the concept of a one-man live show, based on rhythm and involving electronic equipment; and the concept of releasing music through own label, which can be seen as self-publishing (and somewhat similar to the concept of Russian самиздат [samizdat] literature). However, having no means to acquire actual studio equipment (which I'd anyways have found tedious, having already imagined a platform that in a single device would offer me capabilities existing studio equipment didn't), and no starting point to start developing my own DAW, at the time I had to come to terms with temporarily setting away my solo musical career ambitions; and at that time, I came to saw multimedia design as an emerging opportunity, which would allow me creative participation in the music scene while not being a musician - and hopefully, eventually, bring me the means for equipment acquisition or development.

Thus, I came to take the opportunity to study multimedia design in Denmark, and further seek to attend master studies in Medialogy in Aalborg University (AAU) in Copenhagen. These studies, while opening me to the world of software, again led me back in contact with hardware, especially during my internships with the company Brother, Brother & Sons ApS in Copenhagen (where I also did my master's thesis [34]), and my duties as lecturer and supervisor in AAU. In addition, the studies exposed me to a world of electronic music

instrument development, with related academic literature and corresponding conferences such as NIME, ICMC or DAFX. This motivated me to revisit the shelved idea about the development of a DAW for live performance; however, in spite of the plethora of ideas I was exposed to, the only addition from this period to the idea of the platform as such, is the addition of a haptic actuator to the faders. Essentially, the haptic actuator would be a linear motion motor (like a magnetic loudspeaker), which would be small enough to fit in a fader knob, and strong enough so that when fed with audio signal, it would cause a rhythmical haptic sensation on the fingers holding the knob (similar to the haptic sensation felt when a playing loudspeaker is touched). As such, it would have provided a possibility for monitoring alternative to (and independent from) headphones, of what individual channels are playing - even when they are audibly muted from the main mix.

That is why, upon getting the opportunity for PhD studies in Medialogy in 2007, I had to insist that the main topic concerns the design and implementation of the envisioned DAW; after all, that would be probably the only opportunity I'll ever have at developing audio hardware like that - considering my chances for getting employed at the likes of Roland or Yamaha, and having a first-hand experience of such development, are rather slim. As an initial guideline into this development, I produced an initial 3D model of the DAW, a screenshot of which is shown on Fig 2.7.

Fig. 2.7: A model of the planned DAW interface

The outline shown on Fig 2.7 shows the influence of the machines I've met throughout the 1990s and early 2000s: the top four channel strips would represent individual channel controls; below them is a step sequencer interface;

followed by a "DJ mixer" interface, consisting of two master/group faders, and a crossfader between the two. To the right there is a percussive pad bank, and on the left a screen-like interface; the internals show that the machine would be built based on PC components, allowing for a computing and networking capability; turntable interfaces were intended to be additionally attached, and as such are not shown on Fig 2.7. In fact, it wasn't too big of a design problem to develop an initial engine for this platform, in the FLOSS audio processing visual programming software Pure Data (PD). However, there was one technical problem with the software approach: it is *not scalable* easily, in terms of adding individual channels, on a single computer: once too many channels are added, latencies and stuttering of audio are to be expected. And this was the main impetus for looking into how could the engine for this DAW platform be implemented *in hardware*. But the biggest impediment here was that, in spite of my education, I still did not have a *practical* model of how digital audio functions: what sort of *actual* hardware parts do you need to implement the functionality of a digital sampler; and how is that different from the hardware a PC uses to reproduce digital audio?

2.10 Reduction to soundcard

Since one of the biggest limitations of early samplers was storage of audio sample data (e.g. the E-mu ESI-32 had a, from today's perspective, mere 1.4 MB floppy drive with 2 MB random access memory (RAM)), I found the understanding of audio hardware in context of a PC more crucial (as a PC typically provides access to large storage like a hard disk). Considering it is the *soundcard* hardware that has the responsibility to reproduce PC audio, I set upon looking up designs of soundcards for study. Having been exposed to successful open-source audio projects such as PD, I fully expected that I would track down a project discussing the hardware and software internals of the working of a soundcard, providing me with a base for further development; unfortunately, after several months of searching, I found nothing that significantly advanced my understanding in practical terms (as in, something that I could implement myself for study). My expectations may have been too high: I expected material discussing hardware to provide basic understanding of the PC bus interface, followed by analysis, schematics and identification of points in the circuit where digital and analog signals can be observed, supplanted by oscilloscope traces, images and possibly videos (on a related webpage); for software, I expected discussion of the effects of code in respect to known hardware, described as previously, as well as pointers to tools for analyzing the software performance - which ultimately became my standards for the papers in part II. Thus, I was faced with a choice: either I continue working on the level of prototyped DAW interface design, but without reaching the level of practical understanding of digital audio to allow me independent further development in terms of scale and performance; or, I scale down the scope of the PhD project to producing a set of works discussing

the operation of soundcards in as generic and as reproducible terms as possible, but with the risk of not reaching the stage of a playable DAW prototype.

I found the second option – the study of a soundcard – more crucial; and thus I asked, and got a permission, for scaling down the scope of the project to this area. Being aware (and *angsty*) that the architectural hardware and software details of soundcards might be considered protected intellectual property (IP), whose disclosure in an open manner might present a legal problem - the decision, that this soundcard study is conducted as open source (in the sense of building soundcard systems to the greatest extent either from basics/"first principles", or from open-source code and designs), was immediate. Finally, after tracking down the design in paper II-A (which is not quite a soundcard), switching to the GNU/Linux operating system Ubuntu, several years of research and submissions (complicated by my lack of formal training in computer science), the rest of the papers in part II were produced. The results of these papers, while not necessarily implementable directly in the envisioned DAW, have however finally provided me with the necessary experience, to evaluate parts on the market that would be appropriate for a possible future prototype implementation of the envisioned platform. And I firmly believe, that if I had access to comparable material at the start of the PhD project - instead of spending 5+ years to study digital audio as implemented in soundcards, it would have taken me mere months to gain these basics (whereupon I could have proceeded to fit them in the context of the DAW); and this didactic aspect, I believe, is the strongest contribution of the papers in part II.

Chapter 3

On live performance paradigms in looped electronic music

This section outlines some of the main aspects of live performance in context of looped electronic music; which, by extension, would cover a wide area of styles within contemporary popular music, such as hip-hop or electronic dance music. The overview is based on affordances of instruments which have been established in concert use already, and from today's perspective can be considered classic - rather than focusing on later developments in audio loops utilization (such as e.g. the Ableton Live [35] software, or related tools). Finally, a platform integrating these affordances for live performance is proposed, whose technical implementation intricacy might justify its designation as a digital audio workstation. The discussion in this section is illustrated by software examples, implemented in Pure Data [36] with GriPD [37] and developed on Ubuntu 11.04 GNU/Linux OS, and released as open-source software collection named `sd-phdemo` through the webpage associated to this thesis [38].

Specifically, here the focus is on contrasting the affordances of a classic drum machine's *step sequencer*; versus the traditional *DJ set*, consisting of two turntables (vinyl record players, alias gramophones) and what is known as a DJ variant of a sound mixer. In terms of related work, for an academic treatment of the instrumentalist approach to the DJ set (also known as turntablism), the reader is referred to Falkenberg Hansen (2010, [32]) (or for an alternative perspective, [39]); and while the step sequencer in newer academic literature is typically taken as a pre-existing design metaphor [40] and used in novel developments (e.g. [41, 42, 43]), its incarnation in classic drum machines can be found discussed in literature e.g. by Russ (2012, [44]) or Vail (2014, [45]).

It is important to note that both of these systems originated as products, not primarily intended as instruments for live performance of music: while a drum machine sequencer can be seen as designed for non real-time composition of music - a turntable would originally have been designed as a music reproduction

tool, unrelated to music composition as such. Therefore, it would be prudent to specify what is meant by *live* musical performance in this context: it is the act of the performer changing parameters of the sound produced by an instrument, through *real time* interaction, such that musical meaning (melodic or rhythmical) is conveyed to the audience; *and* where there is a possibility of the performer committing an *error* in conveyance of musical meaning, detectable by the audience in a negative sense. This broad definition thus demands from the performer, a minimum of training and expertise with the instrument; and while encompassing the traditional interaction with acoustic instruments, it also dismisses acts in which there is no possibility of error in terms of realtime reproduction of musical meaning (such as merely pressing a "play" button on an audio player device).

By this definition, however, neither the classic drum machine sequencer, nor the DJ set, are essentially live instruments – as both platforms can, and are intended to, reproduce music independently (though in varying degrees) once started. Hence, the definition of live performance can be extended to encompass real-time modifications of the sound of an already pre-programmed (pre-composed or pre-recorded) music – which, the possibility of error notwithstanding, would result with conveyance of *additional* musical meaning, compared to the machines playing standalone. A critical element in the establishment of musical meaning in this scope, is the coupling between the instrumentalist and the audience during a performance; namely, the actual intentions of the performer ultimately do not matter as much, as what the audience *interprets* to be the meaning conveyed by the performance. In this respect, the audience relies as much on visual cues as on auditory ones, in evaluating a live performance: thus, the broad spectrum of body motions of traditional musicians – a drummer playing (for a formal inquiry, refer to Dahl (2005, [46])), or the gestures of an orchestra conductor (for application in music technology, consider Fabiani (2011, [47])) – are much more natural to interpret as a live musical performance, than the minuscule motions involved with, say, pressing a keyboard button.

The distinction in perceiving body motions as live performance is, of course, not an immutable discrete boundary, but rather a continuum - after all, interaction with diverse traditional musical instruments, such as the piano or the trumpet, can be in a mechanical sense reduced to the act of pressing buttons. However, if the evolution of live performance in the electronic dance and hip-hop scenes is anything to go by, it shows that audiences can react quite well to the motions stemming from manipulating linear faders and rotary knobs; motions which are naturally emphasized when the performer is in a standing pose on stage (e.g. twisting a rotary knob may involve not just the palm, but also extend to the elbow of the performer as well, resulting in a more obvious body motion from the audience's perspective). On the other hand, the possibility for performance error emerges from the fact that these devices would already play a pre-programmed sound loop - and in that, at the least impose a rhythm due to the time period of the loop; the performers' challenge is then to stay

in *takt* with this underlying rhythm during their real-time modifications, as audiences typically react negatively to obvious beat mismatching. Therefore, the emphasis here is on the inquiry into the step sequencer and DJ set platforms as tactile, or tangible, musical interfaces: i.e., that which can be achieved in terms of real-time modification of sound, primarily through manual interaction with rotary knobs and linear sliders (but also including buttons) as mechanical, tactile user interface elements.

The text will also outline some specifics of the underlying audio engines, and thus follow the following convention, due to the unfortunate dual meaning of the word 'sample' (noted also in e.g. [48]) in this context: in signal processing, it refers to a single numeric value (obtained for instance as a result of ADC conversion at a particular sampling rate), and the text will use "audio sample" or "signal sample" to refer to this concept; in electronic music, it refers to a short snippet (which is essentially a collection of multiple "signal samples") of sound or music obtained from a recording, which the text will refer to as a "sound sample" (or as per Puckette (2007, [49]), a "wavetable"). Even if the terms "audio" and "sound" are essentially synonyms in everyday use in English, there should be less ambiguation when contrasting "audio sample" and "sound sample" in this purview.

3.1 The classic rhythm/drum machine step sequencer

While there are views, e.g. Arditi (2014, [50]), that the rhythm, or drum, machine historically evolved from a strictly utilitarian need (e.g. to replace studio musicians), the need of musicians to exercise control over - in this case - sound, may also play a large role in this development; otherwise, no one would even want to compose music, having been satisfied with "mere" ecological sound [51]: that which is heard randomly in everyday life. Even if the origins of the drum machine lie earlier in history [52], this section takes four commercial drum machines as a starting point: Roland TR-808 (1980), E-mu SP-1200 (1987), Yamaha RY30 (1991), and Akai MPC2000 (1997), shown on Fig. 3.1 - which have been, and are still, recognized by generations of music producers. From both the user interface and technological implementation perspectives, all of them have certain specifics unique to each device; possibly the most pronounced characteristic unifying them, being the feature of some form of a step sequencer for programming rhythms - and the ability to play pre-programmed rhythms, without human intervention, as standalone instruments.

Perhaps it is more straightforward to start with describing the specifics of the audio engines of these machines, as per the electronic music view of the distinction between audio control interface, and audio generating engine, in an instrument. The Roland TR-808 uses analog electronics circuitry [54], such as voltage controlled oscillators (VCOs), voltage controlled amplifiers (VCAs), voltage controlled filters (VCFs) or noise circuits to produce its percussion sounds. The rest of the machines on Fig. 3.1 all utilize sound sampling, but to a

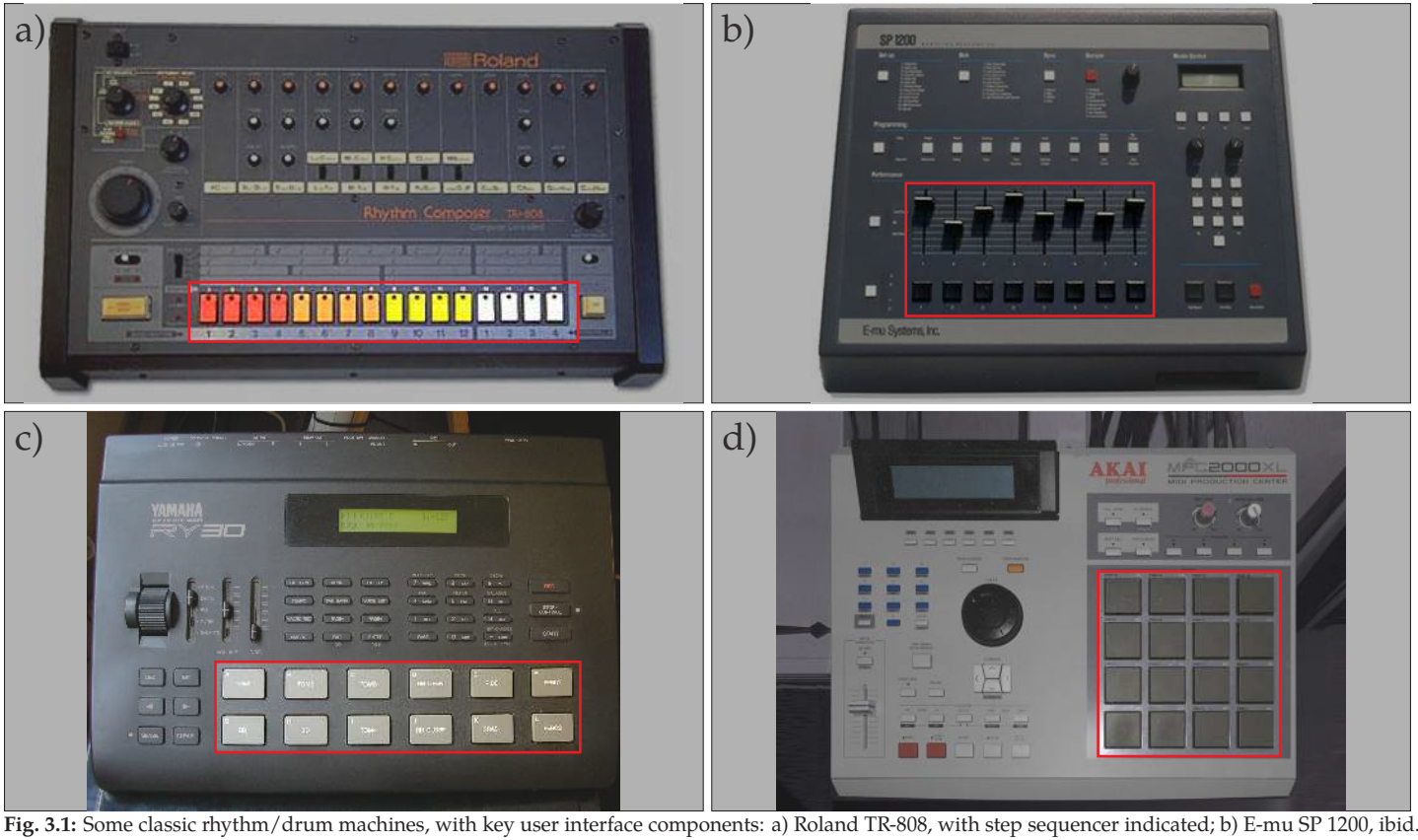


Fig. 3.1: Some classic rhythm/drum machines, with key user interface components: a) Roland TR-808, with step sequencer indicated; b) E-mu SP 1200, ibid.; c) Yamaha RY30, with "pad bank" indicated; d) Akai MPC2000, ibid. (from ref. [53])

different degree: the Yamaha RY30 offers a pre-composed set of 16 bit, 48 kHz sound samples stored into read-only memorys (ROMs), which thus cannot be changed by the user - but could be extended with the purchase of a ROM data card; an approach that may have been known as "sampling & synthesis" or S&S [55, 44] at the time. The E-mu SP 1200 and the Akai MPC2000 are, on the other hand, de facto sound samplers, since they allow the users to record their own sound samples for use as drum sounds: with digital sampling parameters of 12 bit, 26.04 kHz for the SP 1200, and 16 bit, 44.1 kHz with stereo capability for the MPC2000. This difference in engines is, however, no obstacle to the machines sharing user interaction techniques in common, such as those related to the sequencer control interface.

The facilities found in these machines for programming rhythms, can be said to be based in Western musical composition tradition. A discussion on this relationship, from a basic perspective, is provided in appendix A – which is then used to formulate the unique facilities these devices afford, in terms of live musical performance interaction. These unique facilities, otherwise desirable on a generic sequencer interface, are noted as: a button row step sequence controller (as on a TR-808), with a choice of a "note lane" (the step sequencer applies to) through a rotary control wheel (as on a RY30), with individual drum sound "note lane" volumes controllable through a set of mixer faders (as on a SP 1200), and drum pads for real-time playback and sequencing (as on an MPC2000, RY30). Some of these facilities are illustrated in a part of the demo, related to this thesis, released as `seqinterface_s.pd` [38] (graphical user interface (GUI) elements shown also on Fig. 3.5; note that its GUI doesn't reconstruct these particular machines' interfaces, but instead uses a standard 2D piano roll grid, as in typical MIDI editor software).

3.2 The classic DJ set - two turntables and a mixer

The specific affordances for real-time performance in the DJ set stem mainly from the turntable (gramophone, phonograph) device. Some turntables have properties, like heavy weight (increased stability), direct drive (faster reaching of rotation speeds), or pitch control, which makes them especially suitable for DJ tasks - such as the classic SL series of turntables of the Technics brand, that ceased production in 2010 [56]; one example is the Technics SL-1200, shown on Fig. 3.2 left. The DJ set typically consists of two such turntables connected to a DJ mixer, shown on Fig. 3.3.

In terms of audio generating capabilities, the turntable depends on a vinyl record medium, whose content the turntable reproduces. An important consideration here is that the vinyl is engraved with the analog signal representation of the sound, shown on Fig. 3.2 right: commonly, for monaural reproduction, the groove modulation is lateral (also called radial); while for stereo records, the groove cut encodes one channel as radial displacement of, and the other as the width of, the groove [58] (there are also quadraphonic formats, but are not

Fig. 3.2: Left: a turntable historically considered appropriate for DJ sets, Technics SL-1200 (ref. [53]); right: electron micrograph of phonograph vinyl record grooves (ref. [57])

commonly used). That means that a song must be mixed (or downmixed) from multiple individual instrument (or track) channels, to one (for mono) or two (for stereo) audio signal channels. And this, in turn, means that once the song is downmixed, it is impossible to perfectly recover the individual instruments back into separate audio channels. Essentially, the process of mixing is the process of signal addition (summing) in the time domain; if $i_1(t)$ and $i_2(t)$ are two instrument (or simply sound) track mono signals (representing a multi-track mix), then their mono mix $a_{mm}(t)$ can be described with Eq. 3.2.1 (where k_1 and k_2 are mixing coefficient constants).

$$k_1 \cdot i_1(t) + k_2 \cdot i_2(t) = \sum_{x=1}^{2} k_x \cdot i_x(t) = a_{mm}(t) \tag{3.2.1}$$

Even if we simplify, and know a priori how many instrument tracks there are in the mix, and that the instrument signals are in the range of values from 0 to 1, and that the coefficients $k_{1/2}$ are set to 1.0 – we cannot really say what the instrument signals would be, having just the measurement of the mixed signal: if at some moment t_1 we measured $a_{mm}(t_1)$ to be 0.5, we cannot really say if $i_1(t_1)$ was 0.1 and $i_2(t_1)$ was 0.4, or if $i_1(t_1)$ was 0.3 and $i_2(t_1)$ was 0.2 (and for real numbers, there is an infinite amount of combinations). Thus, the recovery of the separate instrument signals from the mixed signal, is undefined even in this simplified mono case:

$$a_{mm}(t) \xrightarrow{?} i_1(t) + i_2(t) \tag{3.2.2}$$

While there is research into algorithms for mix decomposition (i.e. audio source separation) into instrument source signals – both non-informed (i.e. blind source separation [59]), and with prior knowledge (e.g. informed by the notation score [60]), of the characteristics of the mixed signal – such algorithms cannot recover the source instrument signals ideally; are not guaranteed to work in real-time; and their performance may depend on the musical content of

mixed signal itself. As such, they usually can not afford the same convenience as e.g. the SP 1200 interface on Fig. 3.1b does, in terms of real-time control of individual instruments in the mix.

Likewise, the turntable doesn't afford for changing of the tempo of a music composition recorded on vinyl. However, vinyl records are cut at a predetermined disc rotation speed, and their audio content is reproduced in the original manner only when the record disk spins at the exact same speed on the turntable. Several recording speeds have emerged as standard in the industry [58], expressed in units of rounds per minute (RPM), such as 45 RPM and $33^1/_3$ RPM (usually referred to as 33 RPM). By analogy with physics, the symbol for angular velocity ω can be used to denote rotational speed; note that in physics, ω is expressed in units of radians per second $[\frac{\text{rad}}{\text{s}}]$, which has a direct relationship to the (temporal) frequency f in units of [Hz]:

$$\omega = 2 \cdot \pi \cdot f \tag{3.2.3}$$

If one "round" is equivalent to the angle of one revolution, 2π radians, then the conversion factor between rotational speed in RPMs ω_{RPM}, and the rotational speed in radians per second ω_{rps}, can be found through dimensional analysis as in Eq. 3.2.4:

$$1\,[\text{RPM}] = \frac{1\,[\text{round}]}{1\,[\text{minute}]} = \frac{2\pi\,[\text{radians}]}{60\,[\text{seconds}]} = \frac{\pi}{30}\,[\frac{\text{rad}}{\text{s}}] \tag{3.2.4}$$

And the relationship between ω_{RPM} and ω_{rps} can be expressed as:

$$\omega_{\text{rps}} = \frac{\pi}{30} \cdot \omega_{\text{RPM}} \tag{3.2.5}$$

Regardless of the unit, the rotational speed can be considered as a continuous, "analog" function of time, $\omega(t)$. This is of importance, because the rotational speed of the vinyl disc, relative to the turntable stylus, is what determines the reproduction speed of the audio encoded in the vinyl grooves; and thus brings about the most distinctive feature of the turntable (although shared with other analog media, such as the magnetic tape) - the ability to *continuously* slow down, speed up, or reverse audio playback. Notably, the reverse playback occurs when $\omega(t) < 0$ - that is, when the rotational speed is negative (in the sense of being opposite from the default direction of turntable rotation, which is clockwise). This phenomenon is what gives rise to the technique of DJ vinyl scratching [32], as the users can manipulate the surface of the spinning vinyl on a turntable directly with their hands, and thereby directly change $\omega(t)$ (and thus change the audio reproduction speed in real-time). Its artistic potential has been recognized by e.g. Schaeffer (2004, [61]), one of the pioneers of turntable instrument use: "Assuming that we limit ourselves to a single recording, we can still read the latter more or less quickly, more or less loudly, or even cut it into pieces, thereby presenting the listener with several versions of what was originally a unique event" [61].

Notably, changing the reproduction speed of audio, changes *both* the perceived pitch, *and* the perceived tempo (if the recording is rhythmical to begin with), simultaneously. This effect is especially apparent if the record is manually spun back and forth continuously; and is so distinctive, it may have influenced the introduction of scratching in the hip-hop genre (refer to the DJ Grand Wizard Theodore anecdote [62], via [32]). The same effect is also present in the domain of digital audio sampling; in e.g. [49] it might be referred to as transposing a (sound) sample, or the "chipmunk" effect. Note that with digital sampling, it is possible to process audio to change these parameters separately: changing the pitch while leaving the tempo (or the sound sample duration) unchanged is commonly known as "pitch shifting" or "pitch transposition" (e.g. [63]), while the changing the tempo (or the sound sample duration) while leaving the pitch unchanged may be known as "time scaling", "time stretching" or "time compression/expansion" (e.g. [64]); both of these can be addressed by the phase vocoder technique [65].

In spite of the distinctive sound, scratching – as manual transposition of the sound of a vinyl record on a turntable, by changing its relative rotational speed $\omega(t)$ – is usually not musically interesting for long on its own (for instance, as performed on a single turntable). That is why the typical DJ set employs (at least) two turntables and a DJ mixer, as on Fig. 3.3. In the most basic exercise, the DJ set allows for scratching on one turntable, while the other turntable plays a musical recording, most often containing a rhythmical beat. In this sense, live performance on a DJ set can again be seen as rhythmical interventions into a pre-programmed piece of music, thus not unlike the case discussed in the previous subsection 3.1. In fact, in genres like EDM and hip-hop, the beat of the pre-programmed music is very likely to have been produced on machines like those addressed in subsection 3.1 to begin with.

While the individual instrument source signals cannot be recovered from the mixed signal once the vinyl record is cut, the main output from each turntable (usually stereo) in a DJ set can be considered a (meta) instrument signal of its own; and the DJ mixer in the set is intended to address their mixing. Because of that, the DJ mixer usually has some specific characteristics: for instance, a single channel fader usually handles two analog audio signals at the same time: the left and right signal of a stereo mix. Formally, this operation can be described as on Eq. 3.2.6; where $F_{D1}(x)$ is the fader value, in the range from 0 to 1 ratiometrically related to the fader head displacement x (or more accurately $x(t)$, as the displacement can change in time); $l(t)$ and $r(t)$ are the left and right audio signals, where $PF1$ and $RA1$ refer to "post fader" (output) and "raw audio" (input) respectively; and where the index 1 refers to the channel (as seen from the DJ mixer's perspective).

$$\begin{cases} l_{PF1}(t) = F_{D1}(x) \cdot l_{RA1}(t) \\ r_{PF1}(t) = F_{D1}(x) \cdot r_{RA1}(t) \end{cases} \tag{3.2.6}$$

Another distinctive feature of the DJ mixer is the presence of a *crossfader*,

typically identified by its horizontal orientation (Fig. 3.3), as opposed to the vertical one of the channel sliders. The role of the crossfader is to ensure a smooth transition from the music playing on one turntable to the other. Its operation can be formally described as on Eq. 3.2.7; where the signals l_{CF}, r_{CF} refer to output from the cross fader.

$$\begin{cases} l_{CF}(t) = CF(x) \cdot l_{PF1}(t) + CF(1-x) \cdot l_{PF2}(t) \\ r_{CF}(t) = CF(x) \cdot r_{PF1}(t) + CF(1-x) \cdot r_{PF2}(t) \end{cases} \tag{3.2.7}$$

Note that in $CF(x)$ in Eq. 3.2.7 should be seen as a function of the crossfader displacement, which may be linear, but most commonly represents a logarithmic curve known as "crossfader curve", which in some mixers is tunable [66] (main channel faders also typically have logarithmic responses). Furthermore, the displacement x is taken to be relative: with 0 describing one end of the fader; and 1 describing the other - and corresponding to the physical run (length) of the fader. If it is additionally specified in Eq. 3.2.7 that $CF(0) = 0$ and $CF(1) = 1$, then: for crossfader displacement $x = 0$ only the second channel is on the output; for $x = 1$ only the first channel; – and for $x = 0.5$ (midway), it is often desired that both channels are present on the output with equal volume (e.g. $CF(0.5) = 0.5$). Finally, DJ mixers typically include a set of rotary knobs associated to each channel (forming a vertical "channel strip" with the main channel fader), which represent bands in an audio frequency response equalizer (EQ); these are used to great effect in live performance, to rhythmically either amplify or attenuate the "bass", "treble" and "high" range in the audio spectrum.

Thus, the main features of the DJ set in respect to live performance, beyond the real-time possibility to start and stop music, are: the possibility to change sound transposition (pitch and tempo/speed) manually, by manipulating the rotation of the vinyl record played on a turntable; and the possibility to rhythmically intervene into the volume and spectrum of individual songs in the mix through the associated sliders (both channel faders and crossfader) and rotary knobs (mainly for EQ) on a DJ mixer. This represents only the basic, most commonly used facilities; techniques integrating other user interface elements (e.g. "needle drop", manipulating the pitch slider on a turntable) can also be integrated in a live DJ performance.

Some of these facilities are illustrated in a part of the demo, related to this thesis, released as `turntable_audioloop_dbl_s.pd` [38] (GUI elements shown also on Fig. 3.5). The demo features two real-time turntable simulators drawn on the GUI, along with a set of sliders representing main faders and crossfader of a mixer. While capable of loading arbitrary song audio, this demo initially plays through one-measure loops of mixed sound of sequences, implemented in the demo of the previous subsection 3.1 – in order to emphasize the difference in affordances for live performance between the two classic cases.

Fig. 3.3: A DJ set of two turntables and a mixer (ref. [53]).

Fig. 3.4: Manual live interaction with a classic, tactile mixer interface (ref. [67], 15:03).

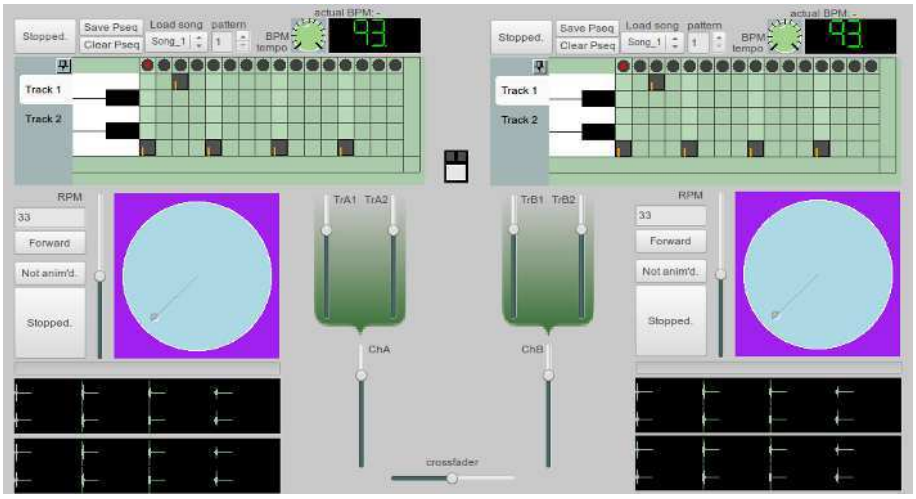


Fig. 3.5: A screenshot of the GUI of the `turntable_seqinterface_dbl` part of the demo associated to this thesis.

3.3 Proposals for user interface facilities merging

The previous subsections on classic sequencers and the classic DJ set, allow for a comparison of their distinctive user interface facilities under a common purview – which may promote a later derivation of a unified user interface, that supports the affordances of both. In general terms, the classic sequencers allow for control of a composition, in granularity from the level of a note, up to (and including) the level of a song; conversely, the classic DJ set allows for a treatment where the lowest level of granularity is a song (and the high level would commonly be called a "mix").

In more specific user interface terms, the classic sequencers allow for change of song elements through buttons, sliders, and rotary knobs, of which the most distinct in terms of live performance are: drum/instrument pads (as special case of buttons, for real-time recording of sequences), step sequencer buttons (for step entry of sequences), and faders for volume control of individual instrument audio channels. Similarly, the classic DJ sets base their affordances on buttons, sliders, and rotary knobs; however, they allow for changes at the level of entire songs, and their most distinctive features are: the turntable as a rotary controller of the sound transposition (pitch and tempo) of a song, and the use of song channel faders and crossfader to rhythmically control the volume - as well as the use of rotary knobs to rhythmically control the EQ - of a song in the mix.

The most obvious point of similarity between these two platforms is the requirement for a discrete user action to start or stop the automatic, standalone reproduction of sound. As the turntable record starts (and keeps on) spinning in order to reproduce sound, especially in cases of rhythmical music, another obvious relationship emerges: that the rotational speed of the record ω is directly related to - and controls - the tempo of the musical composition being reproduced. Noting that as per Desain and Honing (1993, [68]) it is not always accepted that tempo is a continuous quantity, this mapping can be expressed as on Eq. 3.3.1.

$$\omega(t) \rightarrow M_T(t) \tag{3.3.1}$$

In Eq. 3.3.1, the turntable rotational speed $\omega(t)$ can initially be considered to be in units of RPM, while M_T is again a symbol for the metronome tempo in beats per minute (BPM). At this point, some proposals can be considered on how to implement this kind of relationship, which crosses the classical turntable/sequencer boundary.

3.3.1 A trivial mapping from rotational speed to tempo

Since the notion of tempo is considered here in the domain of sequencers, it is possible to reduce it to the value of some memory register in case of a digital sequencer (and the same metaphor can be stretched to include e.g. circuits based on potentiometers in case of analog sequencers). Thus, the conceptually simplest approach would be to track or acquire the turntable vinyl rotation

speed as a signal, and after ADC (if the signal is analog) and appropriate numeric scaling, simply storing the value of this signal in the tempo register. Acquiring the rotation speed from a turntable as a digital signal (in which case it is considered a *turntable controller* [32]) can be done in different ways, e.g. through: commercial vinyl emulation systems, that utilize special time-encoded vinyls, like Final Scratch (whose development history seems to have ended with legal disputes [69]); commercial sensor based systems such as Tascam TT-M1 (which uses encoder wheels that rotate with the record, similar to an optical mouse [70]); or by mounting potentiometers or rotary encoders on the turntable vinyl surface [32].

Even if a turntable controller is in place, and its rotational speed is propagated as a digital value to the presumed tempo register of the sequencer - as long as the underlying audio engine of the sequencer conforms strictly to the model on Fig. AA.3, a complete simulation of the turntable operation will *not* be possible. The reason for this is that a classic sequencer typically merely triggers its audio engine; and from that point on, the audio engine reproduces sound "forward" in time, regardless if it is an analog oscillator, or a digital sound sampler. In case of an analog oscillator, it is in principle impossible to make it oscillate "backwards" in time, although certain effects can be achieved by reversing the transient times of applied attack, decay, sustain, release (ADSR) envelopes [71]. While a digital sampler in theory is more versatile, those found in classic sequencers would typically be capable of sound sample transposition (changing both pitch and tempo/duration) - but only in the range of reproducing MIDI pitch, which is encoded as a 7-bit unsigned integer (i.e. as non-negative values 0 to 127). Thus, even if the mapping of $\omega(t)$ when the turntable spins in reverse, is made to result with a tempo $M_T(t)$ which is negative, it cannot be applied to individual sounds as negative pitch - as the underlying audio engines would not be able to interpret it correctly.

A consequence of this is that even if the turntable controller is spun backwards, the best one can do in this context is to play back a sequence pattern backwards - while the individual sounds still play forward. While this could be an interesting effect, it doesn't emulate correctly the sonic behavior of turntable vinyl sound when playing backwards. On the other hand, this kind of mapping preserves the original meaning of tempo - as the turntable controller is spun faster or slower than its default speed (e.g. $33^1/_3$ RPM), the sequence itself would play at faster or slower tempo, while the instrument sounds stay unchanged in pitch when triggered. This behavior is illustrated in a part of the demo, related to this thesis, released as `turntable_seqinterface_s.pd` [38]; its GUI elements form a half of the GUI shown on Fig. 3.5. The demo also illustrates another potentially interesting possibility for live interaction in this context: the sequence step indicators are clickable, and allow the user to instantly skip and set the "phase" of the sequence at a particular step, with subsequent steps occurring at times set by the master tempo, as per the spinning turntable controller – allowing for limited real-time control of rhythmical transitions.

3.3.2 A sequence-rendering, double-buffered, mapping from rotational speed to tempo

It is not necessary to remain at the trivial form of mapping of $\omega(t)$ to $M_T(t)$, considering that the domain under consideration contains the sequencer – and thus the note-level description of music. If the concept of a song (which has a limited duration) is approximated to a repeating sound loop (which in principle can repeat endlessly, thus having unlimited duration), then there may be some opportunities for a different mapping implementation. Namely, the duration of the sound loop can be determined by the metronome tempo $M_T(t)$ in BPM, and the number of measures it encompasses; and it determines the time period at which the loop repeats. For the simplest case of a $\frac{4}{4}$ loop a single measure in length, the loop duration in seconds d_l, can be calculated as four times the duration of a quarter note d_q as per Eq. AA.1 – which is summarized in Eq. 3.3.2.

$$d_l = 4 \cdot d_q = 4 \cdot \frac{60}{M_T} = \frac{240}{M_T} \tag{3.3.2}$$

A known loop duration in seconds, also determines the size of the loop's digital representation in bytes, at a known audio sampling frequency, resolution and number of channels – and thus determines the size (or, in terms of arrays, the length) of the buffer memory required to store the digital representation. The buffer length L_B in bytes can be calculated using Eq. 3.3.3, where f_s is the sampling frequency in [Hz], S_S is the sample size in bytes, and N_C is the number of channels.

$$L_B = d_l \cdot f_s \cdot S_S \cdot N_C \tag{3.3.3}$$

Thus, a single $\frac{4}{4}$ measure at 120 BPM has a duration d_l of 2 seconds; and with mono (single channel) sampling parameters of 8 bit, 44.1 kHz would require 88200 bytes of memory.

For a digital sampling sequencer, the participating instrument sounds are digital samples, also limited in duration - and known a priori, in the sense of them having to be stored in some form of memory in order to be used. In this context, it is possible to move away from the model of polyphony in classic sequencers, where multiple note lanes in real-time drive multiple audio generators as on Fig. AA.3, in parallel. Instead, *faster* than real-time mixing of the instrument sounds can be attempted, where the MIDI note pattern of the loop sequence is used as input to an algorithm, which then accesses the stored digital sound samples, and mixes them as appropriate (per sequence data such as velocities or pitch) into a final mono (or stereo) digital audio representation with the duration of the loop, stored in a memory buffer. This process (and its result) can be called audio *rendering*, by analogy with the term commonly used in computer graphics (e.g. [72]). A simplified model of the audio rendering process is shown on Fig. 3.6.

The simplified audio rendering model on Fig. 3.6 shows only a single measure of a single note lane (which implies a single MIDI track), which defines

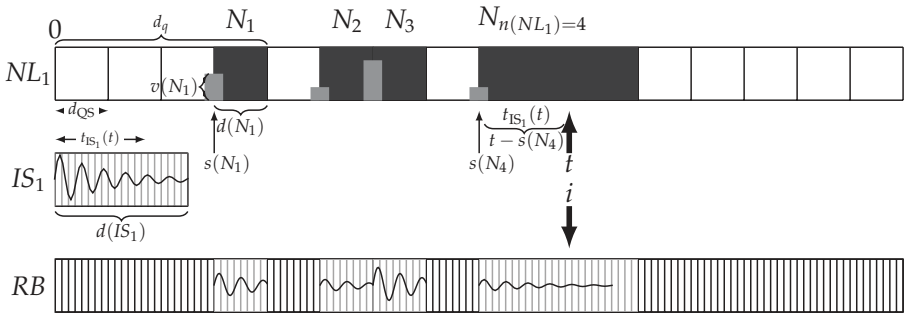


Fig. 3.6: A simple audio rendering model for a single note lane sequence and single associated instrument sound sample.

the sound loop. The measure is quantized in 16$^{\text{th}}$ notes, which determines the duration of a quantum as per quantization settings, d_{QS}; note that the tempo may still be defined in terms of quarter notes, whose duration is d_q. The note lane NL_1 has four notes, N_1 to N_4; a function $n()$ can be assumed, which returns the number of notes for a particular note lane (e.g. $n(NL_1) = 4$). Three more functions can be defined with a note as argument, so: $v(N_k)$ would return the velocity, here in the range 0 to 1; $d(N_k)$ would return the duration; and $s(N_k)$ would return the start of note k in a given note lane sequence. The start and duration can be assumed in units of seconds for the notes on note lane NL_1. Corresponding to this note lane, is an instrument sound, on Fig. 3.6 shown as a sound sample IS_1; it is an array of audio signal samples (for simplicity, here assumed to be equivalent to a byte), stored in a form of memory. Because of this, the duration of this sound sample $d(IS_1)$ is best expressed in units of samples (which is linearly scalable to seconds).

The output rendering buffer RB is an amount of samples in size, which corresponds to the duration of the measure at a given tempo and sampling frequency (the sound sample IS_1 is assumed to have been acquired at the same sampling frequency). The reference point for time $t = 0$ is the left edge of Fig. 3.6; and in the context of the measure, the time t can be considered in seconds – while in the context of the rendering buffer, time maps to the index pointer i in samples. One way to describe the rendering process in these terms is shown on Eq. 3.3.4.

$$RB[i] = \sum_{k=1}^{n(NL_1)} \begin{cases} v(N_k) \cdot IS_1\left[i_{\text{IS}_1}[i]\right] & \text{if } 0 \leq i_{\text{IS}_1}[i] < \min(d(N_k), d(IS_1)) \\ 0 & \text{otherwise} \end{cases} \tag{3.3.4}$$

The buffer RB here is considered as a discrete function of the signal sample index i, $RB[i]$. Note that Eq. 3.3.4 assumes that notes cannot overlap in a note lane; thus, as i (or t) traverses from 0 to end of the measure, it is either within a bounds of some single note, or not (therefore, the sum in Eq. 3.3.4 is used simply as an iteral [73] for k as iterator with explicit bounds; the sum will otherwise

either have only one term non-zero, or all terms zero). The function $i_{\mathrm{IS}_1}[i]$ then returns the index relative to the instrument sample buffer - that is, the difference between the current position and the last crossed start (or trigger) of a note (and has the same meaning as $t_{\mathrm{IS}_1}(t)$ on Fig. 3.6). This also means that upon each trigger, the sound sample phase is reset in this rendering model. The formulation on Eq. 3.3.4 limits $i_{\mathrm{IS}_1}[i]$ to be smaller than the duration of either the given note, or the instrument sound sample; when notes are longer than the instrument sample, Eq. 3.3.4 writes zeroes in the rendering (as Fig. 3.6 implies for N_4). Additionally, the algorithm doesn't perform any additional shaping (e.g. as through an ADSR envelope) of the note events. The buffer-relative index function can be written as on Eq. 3.3.5, where the start and duration functions are assumed to return values in units of signal samples.

$$i_{\mathrm{IS}_1}[i] = \left\{ \begin{array}{lll} 0, & \text{if} & i < s(N_1) \\ i - s(N_k), & \text{if} & s(N_k) \leq i < s(N_k) + d(N_k) \\ 0, & \text{otherwise} & \end{array} \right| \; k = 1, \ldots, n(NL_1) \tag{3.3.5}$$

In order to render the audio of an entire sequencer track, understood as a stack of note lanes related to individual instrument sound (or pitches), let j represent a particular pitch in the track, IS_j the corresponding instrument sound array, and N_{jk} the k^{th} note in the corresponding note lane NL_j. The audio rendering process in Eq. 3.3.4 can now be performed over all j, as on Eq. 3.3.6, in order to obtain a representation of the entire track.

$$RB[i] = \sum_{j=1}^{P} \left(\sum_{k=1}^{n(NL_j)} \begin{cases} v(N_{jk}) \cdot IS_j \left[i_{\mathrm{IS}_j}[i] \right] & \text{if } 0 \leq i_{\mathrm{IS}_j}[i] < \min(d(N_{jk}), d(IS_j)) \\ 0 & \text{otherwise} \end{cases} \right) \tag{3.3.6}$$

Unlike the case in Eq. 3.3.4, here the summation will result with more than one non-zero terms for a particular (time) index i – and as such, the algorithm on Eq. 3.3.6 implicitly performs *mixing* of all the note lanes' content in the track, limited in volume only by the particular note velocity $v(N_{jk})$. The symbol P is used in Eq. 3.3.6 to represent the total number of pitches (note lanes) in the track; in the case of MIDI, this number would be 128, as pitch is encoded with 7 bits. However, in practice, iterations need only be performed over note lanes that actually have notes at a given time into the measure, which can be seen as one optimization strategy when implementing this algorithm in software (on Eq. 3.3.6, this approach can be emphasized if the two sums switch places).

Ultimately, this process can be extended to arbitrary levels of hierarchy nesting; but here, only one more level is relevant: the level of a (MIDI) song, as a collection of (MIDI) tracks. If TR is the iterator over the tracks in a given song, let $\mathrm{MC}(TR)$ return the MIDI channel associated to the track; since the MIDI channel also specifies a set of instrument samples, a particular instrument sample (i.e. its buffer) would then be indexed as $\mathrm{IS}_{\mathrm{MC}(TR)j}$. Additionally, each track may have an associated volume setting (or even a signal, changing in time), which would figure as an additional multiplier in the algorithm equation.

The key result of such processing, however, is the (here, mono) audio rendering present in a single buffer, capturing a measure of the complete sound loop, as described by individual track note-level events. As such, it can be reproduced by only a *single* audio engine of the kind depicted on Fig. AA.3. And that, in turn, means that an available rotational speed signal $\omega(t)$ from a turntable controller, may be applied directly as the playback speed through the render buffer – thus simulating the proper sound transposition (simultaneous change of tempo and pitch) that a vinyl record on a turntable exhibits. The standard playback from an audio sampling generator can be conceptualized through a sawtooth (or piecewise linear ramp) function [49], shown on Fig. 3.7.

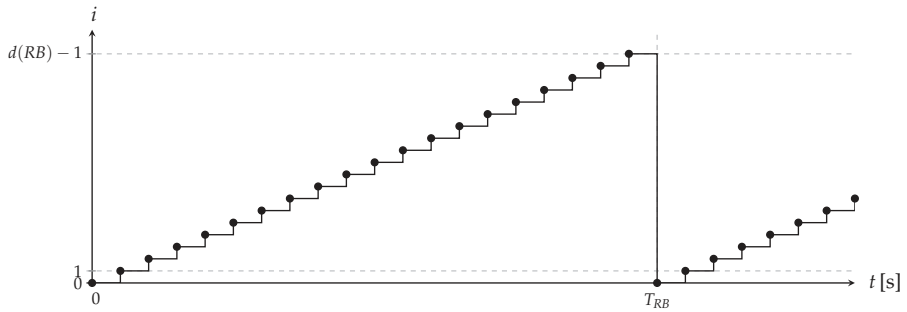


Fig. 3.7: A representation of a "sawtooth" signal, generating the indexes for signal sample playback for a sampler

The sawtooth function on Fig. 3.7 has to traverse all samples in the rendering array, whose amount here is returned by the duration of the buffer $d(RB)$ expressed in signal samples; and has to traverse them in a time period, e.g. T_{RB}, which is associated to the rendering buffer. This sawtooth function can be formally written as on Eq. 3.3.7.

$$i(t) = \left\lfloor \frac{A\frac{1}{T_{RB}} \cdot \mathrm{mod}(t, T_{RB})}{T_s} \right\rfloor \tag{3.3.7}$$

In Eq. 3.3.7, A stands for the amplitude, which here is the total length in samples of the RB rendering buffer array, $d(RB)$; T_{RB} is the duration of the buffer array, same as $d(RB)$ but in seconds; mod is the modulus (modulo) function; $T_s = 1/f_s$ is the sampling period; the independent time variable t can be assumed to increase continuously; however the output $i(t)$ – which is in the same domain as index i used in Eq. 3.3.4 and subsequent – still changes discretely. Thus, if the sampling frequency is $f_s = 44.1\,\mathrm{kHz}$, at which RB contains 2 seconds of audio, then for an identical reproduction, the sawtooth function has to traverse all $A = 88200$ sample indices in $T_{RB} = 2\,\mathrm{s}$. If the sawtooth function traverses the same indices in a time period which is smaller or larger than T_{RB}, the playback speed will be faster or slower accordingly, and sound transposition will occur (pitch and tempo will change); note that in such cases, there are

generally either more or less samples than one per sampling period, and thus some form of linear interpolation between the sample values may be required. Note that Eq. 3.3.7 specifies an "endlessly" repeating sawtooth waveform in time; which as a playback control, in principle, turns a digital audio sampler into a wavetable oscillator [49] – however, here it also ensures that the audio rendering is constantly looped; conversely, if the sawtooth signal doesn't repeat after the first period of T_{RB} expires, it would have been a one-shot sampler.

At this point, a relative rotational speed, $\omega_r(t)$ can be defined as on Eq. 3.3.8.

$$\omega_r(t) = \frac{\omega(t)}{\omega_D} \tag{3.3.8}$$

The relative rotation speed $\omega_r(t)$ has a value of 1 when the actual rotation speed $\omega(t)$ is at some predefined default value ω_D (say, 33 RPM); and is otherwise ratiometrically related to it: an $\omega_r(t)$ value of two, means that $\omega(t)$ is at a value twice bigger than ω_D. This allows the insertion of $\omega_r(t)$ directly as a divisor in Eq. 3.3.7: when the record spins twice as fast as the default speed, the audio rendering in *RB* should be reproduced in half the time period T_{RB}. This is also valid for negative $\omega_r(t)$, whose influence would invert the phase of the sawtooth waveform on Fig. 3.7, which corresponds to reverse playback at the granularity of a signal sample (even if additional scaling may be needed in that case to maintain the proper range). Finally, the turntable rotation speed, in its relative variant $\omega_r(t)$, can be integrated into the audio playback engine of the rendering buffer, which is shown on Eq. 3.3.9.

$$i(t) = \left\lfloor \frac{d(RB)\frac{\omega_r(t)}{T_{RB}} \cdot \mathrm{mod}(t, \frac{T_{RB}}{\omega_r(t)})}{T_s} \right\rfloor \tag{3.3.9}$$

Thus, the rendering process as on Eq. 3.3.6, which fills (or records) the render buffer *RB*; and the process on Eq. 3.3.9, which plays back (or reproduces) the same render buffer *RB* – establishes the real-time mapping between the rotational speed of a turntable controller $\omega(t)$, and the reproduction of a step sequencer pattern at tempo M_T; however, in a manner analogous to the behavior of a vinyl record with a classic turntable.

The playback aspect of this concept can be relatively straightforwardly implemented with modern consumer PC technology, as e.g. the demo referred in subsection 3.2 illustrates. However, insistence on real-time behavior as that of the classic sequencers described in subsection 3.1, essentially means that for *every* action the user undertakes, that changes the sequence pattern (adding or deleting a note, or changing its pitch or velocity) – requires that the *entire* rendering process runs again, for all tracks and all notes in the song, to generate the newest up-to-date audio rendering of the looped sequence. As Eq. 3.3.6 hints, this is not a trivial process, at least in the sense of it requiring time to do its task, which increases with the amount of notes (or generally, event attributes) in the sequence, and the duration of the sequence loop (which increases with the

decrease of metronome tempo). This is especially visible in more elementary implementation approaches of these algorithms in consumer PC software, when a *full-duplex* operation is required: that is, when recording into and playback from the rendering buffer should occur in the same time. Namely, depending on the application programming interface (API) and the OS, sometimes writing into a buffer may completely stop the playback from it for the duration of the write; other times the algorithm can end up being so demanding that the OS may start swapping memory from RAM to hard disk or vice-versa, which can also interrupt a playback operation. At the very least, an implementation of an algorithm that causes loss of audio playback in random intervals, makes it unsuitable for live musician interaction.

This is where the dual or *double buffering* concept comes in, traditionally a concept used in computer graphics [74]. Essentially, that means that in the context on Fig. 3.6, instead of one rendering buffer RB, there will be two such buffers, RB_1 and RB_2. In this context, while one of the buffers (say, RB_1) is in the role of a *front* buffer, i.e. has its data read and used by the playback process – the other, the *back* buffer, would wait until a the user effects a change in the sequence data, that requires the rendering process to run again. The rendering would then proceed to fill the back buffer RB_2 in the background, while the front buffer RB_1 is still playing; and once the rendering process completes, the buffers switch their roles: RB_2 becomes the front buffer, and the source of playback data, while RB_1 becomes the new back buffer. Since the playback process reproduces only one sample at a time, conceptually this process could occur with only a single buffer, without deleterious effects to playback quality - however, the double buffering strategy would provide a workaround for possible mutually exclusive locks on read and write operations in a single buffer (here as memory location), which might be enforced by PC software APIs. Switching from one to another buffer as source of playback, may result with reproduction of consecutive signal samples that differ in value in unrelated manner, and thus might cause an audible pop or click, degrading the playback quality; this can be addressed in different ways, for instance by allowing such transitions to occur only at the start, or at only at each quantization step, of the loop measure – or by implementing a short (few milliseconds) crossfading transition between the two buffers, while allowing the transitions to occur anywhere in the measure (down to the granularity of the sampling period). Ultimately, techniques like these would allow for, roughly speaking, the possibility for a musician to modify or program a loop sequence, and then "scratch" it *immediately* afterwards through a turntable controller, in near real-time – which represents a live performance action, that merges the affordances of both classic drum machine sequencers and classic DJ sets.

Note that the entire discussion so far is for a percussive, "non-pitched" operation: pitch (or its note lane) has been considered an index into a set of instrument samples, which are copied by the rendering process as-is. For a pitched operation, where note lane events utilize one and the same instrument sample but at different pitches, note first the formula [49] for conversion of

MIDI pitch m to (fundamental) frequency f in [Hz], shown on Eq.3.3.10; where a MIDI pitch number of 69 corresponds to the note A above middle C (A4, which is standardized to a frequency of 440 Hz).

$$f = 440 \cdot 2^{(m-69)/12} \tag{3.3.10}$$

In more general form, Eq. 3.3.10 can be rewritten [48] as Eq. 3.3.11, where g is a known frequency of a reference note (say A4's 440 Hz), and a is the (positive or negative) number of equal-tempered semitones away from the reference note.

$$f = g \cdot 2^{a/12} \tag{3.3.11}$$

Since time period is inversely proportional to frequency, the duration of a pitched sample would also inherit the same logarithmic relationship as on Eq. 3.3.11. That is, for pitched operation, an instrument sound sample would have to be played faster or slower by the corresponding factor than its original duration – or, its array would have to be resized, "compressed" or "stretched" in total size in samples accordingly (requiring linear signal sample interpolation), *before* the rendering process (as on Eq. 3.3.6) can correctly place it (as transposed) in the final audio rendering. Also, the discussion so far covered only mono operation; for stereo (or multiple audio channel) operation, there should be a separate double buffer for each output audio channel, and the algorithm needs to operate synchronously on these buffers in parallel.

These facilities are illustrated in a part of the demo, related to this thesis, released as `turntable_seqinterface_dbl_s.pd` [38], whose screenshot shown on Fig. 3.5. The GUI contains two "decks" – each of which consists of a turntable controller and an associated sequencer with own set of instrument samples – whose final output can be controlled by main faders (ChA, ChB) and a crossfader, thus simulating the affordances of the classic DJ set. The demo operates in stereo, 16 bit, 44.1 kHz; and each sequencer contains two tracks of five note lanes, one percussive (non-pitched) and one pitched. Each of these tracks has their own dedicated stereo rendering double buffer (i.e. four buffers in all), in order to allow for smoother response to change of track volume through the track sliders (TrA1, TrA2, TrB1, TrB2). Each change of sequencer data (changes to note, pattern, song, or BPM tempo) cause the affected track buffers to be re-rendered and switched. The turntable controller drives the front buffer playback for each track individually; thus the engine could handle "scratching" on the turntable while at the same time "cutting" the volume of *individual* tracks through the track faders (even if that is impossible to perform with a mouse on this GUI). Since the engine for turntable control is already in place, the waveform images in Fig. 3.5 also serve as a controller alternative to the turntable: as they should in principle visualize the waveform of the final rendered loop, by clicking and dragging in them, the relative horizontal position could be mapped to a position into the loop sequence; thus a linear audio "scrubbing" live action is implemented, as alternative to the rotary "scratching" one (alternatively, a

quick press/release on the image can be used to quickly seek into the respective position in the loop).

The rendering algorithm in the demo is implemented in the Python scripting language, as a compromise between performance and ease of rapid development, using the `py/pyext` [75] extension objects for Pure Data. As such, optimizations like use of the Python package `numpy`'s vectorization techniques, were required to obtain a relatively stable performance on a modest recent PC hardware as development environment (see paper II-G). Among other optimizations, the rendering process doesn't resize the pitched instrument sound samples in their entire duration, as this may take prohibitively long time for long sound samples, especially in the context of chords; instead, the algorithm first attempts to find the minimal amount of note events per quantization step in a track, then iterates through each step, and determines the corresponding range in the source instrument sample, before resizing that snippet only. Even with optimizations like these, the demo (within the constraints of the development environment), can only perform seamlessly under some conditions: e.g. modifying notes in a pattern with few notes present. Changing a sequence pattern may introduce audible, but tolerable silence clicks in the loop playback audio; while changing the tempo (which requires a resize, then re-rendering, of all buffers for all tracks in a song), and changing a song (which implies loading of both a new sequence pattern, and a new set of instrument sound samples, possibly from hard disk) typically cause intolerable silence interruptions of playback.

3.4 Discussion

This section contrasted the affordances of the classic drum machine sequencers and the classic DJ set, in terms of musical live performance, both from an interactive and a technical viewpoint; and proposed methods for their merging - the allowance for their simultaneous manipulation on a digital instrument platform. While the ideas outlined herein might have been novel at the turn of the 21st century, it is difficult to assess a remaining novelty factor a decade and a half later; related though not identical academic approaches can be found in prototypes like e.g.: Andersen (2003, [76]), whose Mixxx software accepts MIDI events from augmented hardware, and assists DJ navigation by visualizing audio; possibly the closest in intention is Lippit (2004, [77]), which submits a system of turntable controller (further elaborated in [70]), foot pedal, custom commercial drum pad-based (16padjoystickcontroller) and joystick based (Lupa) controllers to control audio loops in software, but disregards composition-level events; Pashenkov (2004, [78]), whose Diskotron is an optical turntable controller fed with paper disks, printed with alphanumeric commands, which are interpreted and output as MIDI events; Fukuchi (2007, [79]), which explores multi-track, multi-touch scratching, but does not treat these tracks as commonly bound by a composition parameter such as tempo. Evaluations contrasting the usability of different

approaches of this kind have been performed in e.g. Lopez *et al.* (2011, [80]).

Ultimately, in the system proposed herein, the turntables can be considered as external controllers, and even replaced with different but related controllers (e.g. based on a repurposed hard disk as in Yerkes *et al.* (2010, [81])). The system here relies more on the user interface affordances of not just buttons (also in the form of pads), but also rotary knobs and linear sliders (an evaluation of which can be found in [82]). By now, it would be nearly a standard practice to proceed with e.g. the code already implemented in the presented demos, and otherwise develop a standalone user interface with these tactile elements, packaged in a standalone unit (not unlike the classic drum machines on Fig. 3.1, or the preliminary Fig. 2.7). An initial design layout of such a user interface is shown on Fig. 3.8, which features the controls mentioned as relevant so far: the 16-step sequencer button row with a rotary wheel controller, instrument/drum pads, both individual track and main channel strips (with volume faders and EQ rotary knobs), and a crossfader. Such an interface could provide digital control signals from the user elements to a PC, which would propagate these signals as inputs to the audio engine software, and possibly render a GUI based on them. With a prototype of this kind in place, testing with expert (musician) users could be conducted, to determine the relevance of the approach. In recent years, a usability and experience evaluation methodology, that includes body movement, has evolved for evaluating audio interface devices – both in the domain of digital music instruments, and for other devices like game controllers (consider e.g. Gelineck (2012, [83]) or Böttcher (2014, [84])) – which can serve as basis for the user testing procedure.

The reasons for the categorization of this platform as a DAW, even in this kind of a prototype, are that the algorithms perform real-time mixing of audio in the digital domain, originally found in digital audio mixers like Yamaha 01V (1998); while preserving a separation of track (instrument) audio in the digital domain, originally found in digital audio multi-track mixers and hard-disk recorders like Roland VS880 (1996), to which the term "digital workstation" was originally applied. Furthermore, the algorithms operate in both composition (i.e. note or MIDI level domain) and the digitally sampled audio domain, which is also found in commercial studio software capable of recording digital multitrack audio - to which the term DAW is, and may have been originally, applied to [45]. In fact, it seems that a dedicated approach to multi-track digital audio recording, aside from general studio equipment compatibility, is the only aspect that would separate such a prototype from the concept of a DAW proper (although, as the track renderings are kept separate, the necessary basis for implementing this approach is already in place).

However, the major problem from such a prototype, as the demos clearly demonstrate, would be the issue of performance: musicians would certainly have difficulties relating to an instrument, in which perceivable audio corruption occurs randomly upon (the more technically demanding) user actions. On one hand, for anyone familiar with the studio environment from the turn of the century, when personal computers and digital studio equipment represented

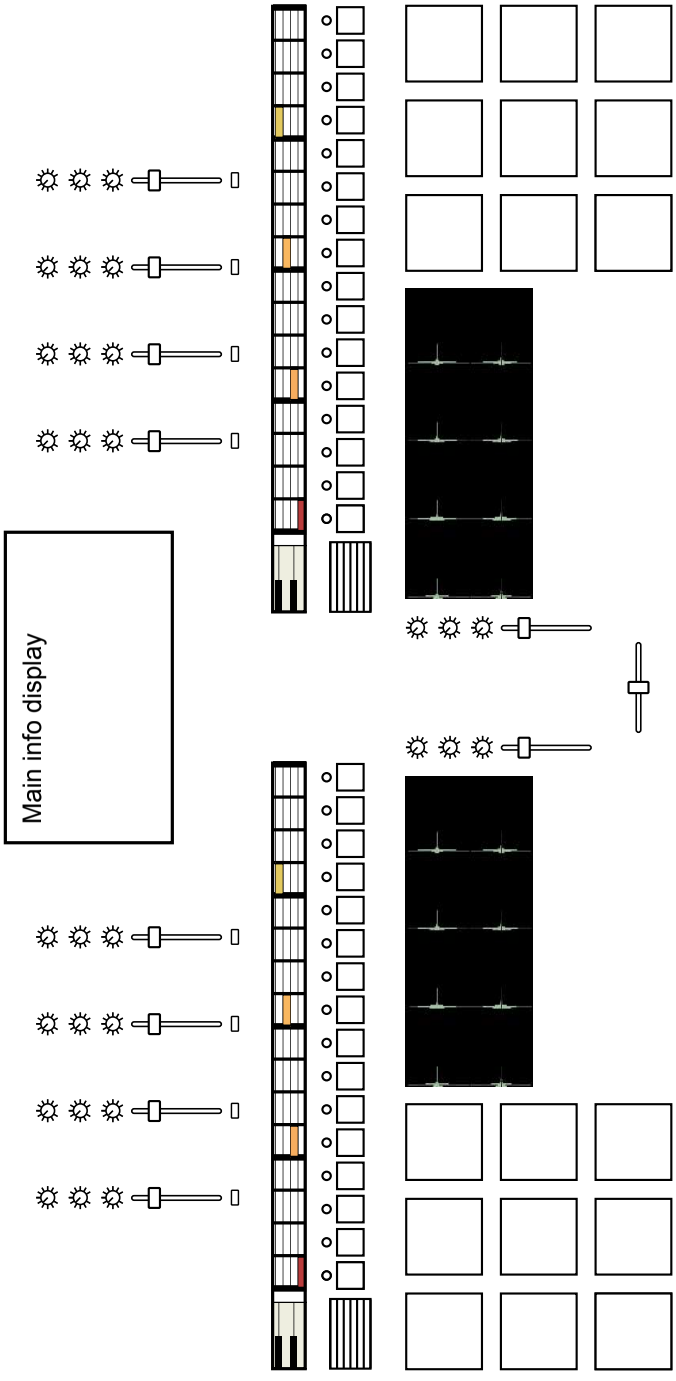


Fig. 3.8: A user interface layout design proposal, with dual 'decks', each of which contain a 16-step sequencer row, 4 track channel strips, drum pads, sequence and waveform feedback screens, along with main mixers and crossfader

related, but distinct, domains – where the studio equipment was more limited in what it can do, but more reliable in what it did, than PC software – the idea of implementing the audio engine *in hardware* would naturally occur. On the other hand, the recent advances and availability of computing power, raise a question in terms of level of detail at which this hardware implementation should be attempted: if the system, say, depends on external storage for saving and loading sound sample data, why go through the trouble of implementing a hardware controller for a 1.44 MB floppy disk - when for relatively modest means it would be possible to integrate a small-factor laptop PC, with built-in hard disk on the order of gigabytes, inside the physical prototype itself?

Thus, if a hardware implementation approach is required, the question is not if, but to what extent - and more importantly, *how* - could the eventual hardware implementation be integrated with a computer. Or more generally, what does hardware implementation *mean* in this context? While a "grounds-up" or "first-principles" perspective is necessary in the scope of this proposed system, it is hardly sufficient in the depths of the hardware design domain: for instance, conceptually it is clear that on this level, the issue of ADC and DAC converters should be considered; yet, introductory engineering literature will most often discuss the devices themselves, without extending to use cases (see paper II-F): thus, for instance in digital audio, it is rarely made apparent that besides ADC and DAC, *also* a power supply, clock signal (setting the sampling frequency), some form of (e.g. RAM) memory, and circuitry for traversing the memory locations index (e.g. a counter) is *required* to even start considering the basic audio reproduction or recording operations. Experienced engineers might claim that "it follows" from theory, however, not everyone has access to the kind of culture where work is done on the specific level to make that apparent (even if the growth of the Internet might make this distinction less important).

In terms of audio, an obvious way to approach this would be to go through a reproducible exercise, or set of exercises, where a practical hardware implementation of circuits is undertaken, which would illustrate both the basic user operations like reproducing high fidelity digital audio, and aspects of interfacing with a consumer PC. Maybe not surprisingly, there are currently very few, if any, examples in literature of projects that aim to be hardware exercises for this particular domain (again, more in paper II-F); one reason for that may be that real world electronics, at least in the consumer area, rarely deals with implementing generic designs – but instead markets proprietary, customized components. And knowing exactly *which* components would be compatible enough with each other, to implement a particular operation reliably, can be seen as a part of a culture rarely present outside the manufacturing business world - where these kinds of choices could be expected to be part of everyday work.

Thus, the lack of practical electronics hardware exercises at the time this project started, that could illustrate generic digital audio issues, either standalone or within the domain of personal computers – and thus serve as a basis

for hardware implementation of the live performance DAW proposed herein, – forced the reduction of this project's scope to identifying, implementing and documenting such examples. When considering digital audio in the context of PCs, there is in reality only one type of a device that immediately stands out as a candidate for study – the *soundcard*. The work involved with the study of the soundcard, the results of which are the papers in part II, is discussed in more detail in the next chapter.

Otherwise, if an engine was in place, that reliably performs audio rendering, by splitting it between dedicated hardware and a computer with plenty of processing power remaining for other tasks, – it would not have been difficult to imagine future perspectives for development of the proposed live performance DAW: interfacing with embedded analog devices like oscillators; haptic feedback for the main and track faders (similar to the D'Groove haptic turntable [85], but utilizing a processed form of the audio that each fader is individually "playing" as the signal source for haptic actuation); usage of networking and related protocols like OSC [86] to allow for collaborative or group performance (e.g. by choosing one of the performers' machines to be a tempo master, to which the others would synchronize); or extensions with further tactile user interface elements relevant to live performance, such as linear "scrubbers" (that can be implemented either through touchscreens, touchpads, or flat membrane potentiometers). The problem of scalability is likely to persist, however: not only in terms of technical solutions, which will naturally impose a limit to, say, how many tracks can the engine process reliably, which the user might want to extend at a later date – but also in terms of user interface. The optimal placement of tactile elements is likely to be different depending on both the capabilities of the system and the individual user preferences; yet, the platform cannot emulate systems for arbitrary placement of tactile UI elements like Villar *et al.* (2005, [87]), because the intended live concert use requires a degree of mechanical sturdiness, and the thus required encasing of the device would "freeze" the tactile UI features for each individual prototype implementation.

Chapter 4

Contributions of the present work: the open soundcard in focus

The previous, chapter 3, illustrates that modern developments in digital electronics and computing had brought about a condition, where ideas in the domain of digital music instrument design are possible to implement as prototypes in consumer technology, that is not only widely available, but is also of the kind that is free and open-source in nature; this is qualitatively different from the conditions historically present at the turn of the century and earlier. However, chapter 3 also illustrates that for some (if not most) such ideas, a development bottleneck may occur at issues of technological performance quality – traditionally, a problem domain in electronics and computer science (software) engineering. Thus, digital lutherie can be seen as a domain, that includes a cross-disciplinary intersection between these two fields. The basis of this intersection – the area of low-level interfacing between software and hardware – is, however, not widely present in the digital lutherie discourse; at least, not in the sense of providing a foundation to the wider researcher community, where a common background is often assumed, but where individuals may otherwise have widely differing experience in either of the engineering fields.

The soundcard, as an ingrained part of now ubiquitous personal computing systems (whatever form they may take), has been identified as a suitable model for study – in particular, practical study as a laboratory exercise – which might address this situation: by demystifying full-duplex, high-fidelity, multi-channel digital audio reproduction and capture, as one of the more involved topics. But what would a practical laboratory study in this context mean? Simply speaking, it means that: in terms of hardware, it would be possible to build device hardware from as generic parts as possible, with clearly identifiable points of measurement of both digital and analog electric signals (which also implies understanding of the measurement tools, e.g. oscilloscopes, and their limits of applicability to this task); in terms of software, code would be written

that interfaces with such device hardware – which implies understanding of the possible interactions of such code, with other constituent software of the operating system it runs on – and measurement of its effects in the functioning of hardware.

This represents the *educational* perspective on the soundcard, covered by the papers submitted in part II, summarized in the next section 4.1. These papers can be seen as a series of practical tutorials, starting from a virtual, software-only soundcard driver; and progressively covering software and hardware implementation issues for up to high-fidelity - stereo, 16 bit, 44.1 kHz - soundcards. As such, while these papers do not necessarily bring about generation of essentially new knowledge (at least not in the engineering fields), they serve a purpose in possibly the next most fundamental academic endeavor: dissemination of existing knowledge. An underlying thesis here is that such efforts would benefit the digital music instrument research community - and eventually bring about better understanding of, and more creative solutions in, this problem domain; whether this turns out to be the case, can only be confirmed by following the developments in the community in the future.

The insistence on an open development approach to soundcards, even if being motivated by economic factors, however also brings about possible new applications in the *research* perspective on the soundcard. The current ubiquity and reliability of the device makes it an often employed part in implementation of prototypes in both media technology research, and in other fields as well, often implicitly (e.g. sonification of physics data or simulations [88]). In media technology research in particular, besides the implicit use of a soundcard to play or record sound, it is often used as a generic data acquisition device, capturing signals from diverse signals and delivering them to software applications – or, as a driver of actuators other than loudspeakers. This perspective is covered by the papers submitted in part III, summarized in section 4.2. These papers represent a sample of diverse areas in media research, with main topics generally within the context of either digital music instruments, or virtual reality. Most of the projects described therein either use commercial soundcards, however with additional workarounds to make them suitable for a data acquisition tasks; or use data acquisition hardware that might have been replaced with high-fidelity soundcards with superior performance, had there not been for incompatibilities typically found in consumer units. An underlying thesis here is that an open soundcard platform would have allowed for cheaper hardware interventions than by modifying consumer soundcards, thus improving research efforts by delivering higher quality data; this is, however, subject to availability of such a platform – open to modifications – to begin with.

These two perspectives, and the respective papers they encompass, are summarized in the subsequent text.

4.1 The soundcard as a didactic model for laboratory exercises in digital audio

In everyday use terms, it is easy even for a lay person to define what a soundcard is: it is a device meant to allow a computer to play or record sound. In more specific terms, it is a device that is meant to allow reproduction or capture of digital audio, as opposed to "just" digital music: a computer capable of MIDI interfacing can play or record music digitally (through external instruments), but that doesn't mean capability of reproducing generic audio, which also includes speech, or any type of noise (ecological, as in field recordings, or not). This straightforward definition is, in fact, a didactic advantage: at least, it allows potential students to easily conceptualize what the *end result* of the exercise should be, which is not always the case with hardware or software exercises: and that is to hear audio being reproduced by the computer. Furthermore, to confirm the end result, no specialized tool or process other than one's moderately healthy hearing is required.

While conceptually simple, embodying this concept as a laboratory exercise is complicated. Unfortunately, one cannot simply go out, and carve a working soundcard out of any rock lying on the street; the soundcard evolved from a utilitarian need to provide personal computers with the ability to reproduce digital audio - and in that, the soundcard also follows the evolution of PCs. Therefore, any actual implementation will first demand a choice of particular commercial hardware and software (or technology in general) with which to build; and as such would necessarily enforce a degree of *vendor lock-in* [89] upon both potential students, and institutions facilitating the laboratory exercises.

Ideally - at least for a software OS - siding with the market share leader, in this case Microsoft Windows, would be an obvious, no-brainer choice. While vendor lock-in doesn't necessarily have to be a problem [90], this market leader has long been known for its proprietary governance of its operating systems' source code, as well as requiring entering into non-disclosure agreements (NDAs) with partners - often with an unsavoury business practice slant, that has triggered interventions of the legal system in the past [91]. Considering that in a soundcard exercise, the binding at the lowest levels of operating system software vis-à-vis hardware will be in focus, it is not difficult to predict problems occurring at that level, whose troubleshooting would require consultation of the internals of the operating system. Microsoft, through its Shared Source [92] initiative, makes the OS source code available, but only to "qualified customers" [93]; and these qualifications, apart from being a Government or an Enterprise, seem to be for an individual to be recognized as a Microsoft MVP (most valued professional) – or in other words, to be a certified member of the guild [94]. A company that goes through all this trouble in protecting their profits, is not likely to look keenly upon the prospect of their operating system internals being publicly revealed, even if it is done under the auspices of education. Thus, the choice of an open operating system – with unfettered access to its source code

– for implementing soundcards as laboratory exercises, seems to be the more reasonable choice; not the least, in order to avoid potential legal entanglements upon publishing details of the same exercises.

In terms of free & open operating systems, it is important to note that the domain of interaction between hardware and software is reserved for the inner core of the OS, known as a *kernel*. There are currently two major open source kernels [95]: the Berkeley Software Distribution (BSD) derived ones (such as OpenBSD or FreeBSD); and Linux. Due to the larger amount of written resources on the Internet in relation to Linux, and especially its audio subsystem, Advanced Linux Sound Architecture (ALSA), the Linux kernel ended up as the choice for implementation base of most of the soundcard exercises. The kernel was utilized as a part of the Ubuntu GNU/Linux distribution, specifically the Desktop edition of versions 10.04 (Lucid Lynx) and 11.04 (Natty Narwhal) – both of which are built on 2.6.x Linux kernel version series. Apart from paper II-A, all papers in part II are set in this operating system environment – and an overview of the technologies studied therein is given on table 4.1.

Project	Hardware	Software	PC bus interface	Audio quality
Extending... *paper II-A*	Custom board	User-space C	ISA (parallel)	8 bit/12.7 kHz mono
Minivosc *paper II-B*	(none)	OS driver	(none/virtual)	8 bit/8 kHz mono
AudioArduino *paper II-C*	Arduino board	OS driver	USB/serial	8 bit/44.1 kHz mono
AudioArduino analog board *paper II-D*	Custom board (+ Arduino board)	OS driver (AudioArduino)	USB/serial (AudioArduino)	8 bit/44.1 kHz mono
Audio bare-bones FPGA *paper II-E*	Custom board	OS driver	USB/parallel	8 bit/44.1 kHz mono
Open s.card as lab platform *paper II-F*	N/A	N/A	N/A	N/A
Comparing CD timing... *paper II-G*	(none), Arduino board, Intel HDA	OS drivers	(none/virtual), USB/serial, PCI	16 bit/44.1 kHz stereo

Table 4.1: Comparison of technologies used in the open soundcard papers in part II

As table 4.1 outlines, the papers in part II can generally be seen as a series (albeit not chronological, see table 1.1) of tutorials, discussing different aspects of software and hardware of digital audio in increasing quality (and thus performance demands): from mono/8 bit/8 kHz, to CD quality - stereo/16 bit/44.1 kHz. Note that paper II-F is an overview of papers II-A to II-E with an emphasis on

their educational perspective, and as such does not deal directly with technological issues as the others on table 4.1.

Operation at CD quality is considered an important goalpost for two reasons: first, because the introduction of consumer soundcards (e.g. Sound Blaster 16) capable of operation at that quality, was the key event changing the public perception of desktop computers into devices capable of *hi-fi* (high-fidelity) audio reproduction, both music and speech; second, because it represents a minimal implementation of both a multichannel (carrying only two channels, the left and the right, in parallel) and a full-duplex (simultaneous playback and recording) audio system. As such, it is worth noting that the specific CD quality sampling parameters, 16 bit/44.1 kHz, are de facto standards as per the 'Red Book' specification for Compact Disc Digital Audio (CD-DA) by the companies Phillips and Sony, later ratified as IEC 60908; note that this standard is not freely available, as it requires licensing against a payment and signing of a confidentiality agreement [96] - however, it is also mostly concerned with parameters of physical manufacture of compact discs. Otherwise, these specific parameters have been inherited from properties of home video tapes used historically as media for digital audio [97, 98], and have emerged [99] based on physiological studies into human audible bandwidth, that agreed on an upper cut-off frequency at 20 kHz (thereafter, the Nyquist theorem would demand sampling at a frequency of minimum twice the upper boundary of the signal spectrum, in order to have a chance at ideally reconstructing the analog signal in the time domain).

Since a more technical outline of the first five papers is provided in paper II-F, here the main contributions can generally be summarized as:

- A webpage, associated to each paper, on the project website [38]; serving as an online appendix with:
  - Additional building instructions and errata
  - Released source code and schematics files
  - Media (images, videos) and data (e.g. oscilloscope captures) files
- Focus on a free and open toolchain for reproducible building of the examples:
  - Schematics and printed circuit board (PCB) layouts for boards (released for papers II-A, II-D and II-E) have been implemented in the free, open-source and cross-platform KiCad EDA software suite
  - Most of the hardware discussed - both the released designs, and the Arduino Duemilanove board used in papers II-B, II-G - can be rebuilt with hobbyist means (i.e. cheaply and individually)
  - All of the software released - driver or user-space code - can be compiled and executed in a free, open-source GNU/Linux operating system; and is released under the same terms
  - All of the software has been developed, tested and described in respect to "vanilla" (official) distributor releases of the kernels used;

thus specialized (e.g. "realtime") kernels are not required
  - Some of the released software represents independent tools, meant to facilitate inspection of different aspects of the soundcard system operation (e.g. the `numStepCsvLogVis` software in paper II-G)
- Practical and contextual illustration of hardware and software computing issues:
  - The distinctions between parallel (ISA, PCI; papers II-A, II-G) and serial (USB, RS232; papers II-A to II-G) data communication protocols
  - The specifics of digital to analog conversion of audio signals, and use of PWM in this context (papers II-C, II-D)
  - The differences between a microcontroller-based (paper II-C) and raw digital logic based (paper II-E) approaches to full-duplex soundcard hardware design and implementation
  - The influence of details like: the phase between the playback and capture periodic kernel callback functions; or, the technique of direct memory access (DMA); on the full-duplex performance of an ALSA driver (paper II-G)

All of the papers and related implementations consider only exchange of uncompressed pulse-code modulation (PCM) audio data; with that baseline, it would be relatively straightforward to extend the exercises with data compression, or real-time audio processing, schemes - on either OS driver, or hardware, level. This development approach allows, for instance, a generic conceptual model of a soundcard to be recognized, shown on Fig. 4.1, consisting of a: bus interface, through which streams of audio data are exchanged with a PC; a binary (crystal oscillator) clock, which ideally would provide both a rate for digital logic (i.e. computing) functions, and the audio rate for handling ADC and DAC conversion; on-board memory, capable of buffering potential bursts in the streams exchanged with the PC; and channel demultiplexer, propagating data between the on-board memory and the respective ADC or DAC units. Most of these units might be physically housed in the same IC chip (as in both paper II-C and II-E); but their implementation details will differ significantly depending on the technology (microcontroller and field-programmable gate array (FPGA), respectively).
Interestingly, this kind of architecture allowed the project to steer clear of deeper involvement with OS kernel issues, until the quality demands reached CD quality – simply because at lower performance rates, deviations from perfect periodic operations on the OS level, are masked by the presence of the on-board buffer memory and an independent audio rate clock. At CD quality, this is no longer the case, and the details behind this behavior, are investigated in paper II-G. Paper II-G notes an unobvious behavior: even when there is USB serial bandwidth of say 200 kB/s, a transfer of CD quality stream which requires 176.4 kB/s may still result with errors: essentially random delays due to kernel pre-emption may influence the queuing of data, resulting with buffer overflows on the soundcard buffer as per Fig. 4.1 - even if the OS doesn't detect an error in that case.

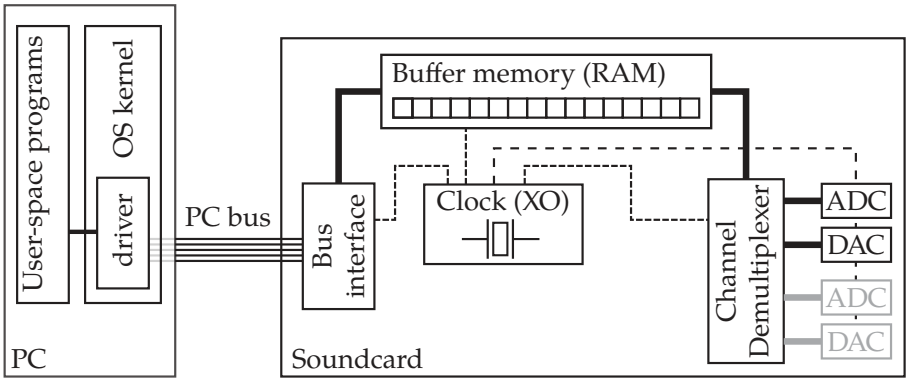


Fig. 4.1: Generic model of a soundcard. On the Soundcard block, thick lines indicate audio data paths, dashed lines indicate different clock domains' signals (additional circuitry performing the control logic based on clock signals is not shown; neither is power supply). A pair of ADC and DAC can be considered as a single mono full-duplex channel.

Paper II-G doesn't provide a solution to this, just states it as a reproducible problem; one approach would be to "shape" the traffic speed according to the actual buffer pointers – but in this case, those buffers are on the FTDI USB-serial chip, and it seems the information on whether such pointers are readable by the PC (and how) is covered by an NDA. Otherwise, another approach would be to implement a compressing coder-decoder (codec); the papers in part II deal strictly with uncompressed audio, since more straightforward schemes like DPCM (differential pulse-code modulation) can be left as exercise to the reader, while other approaches (e.g. MP3 [100]) may be patent encumbered. Still, note that the FTDI USB-serial chip is not capable of isochronous data transfer type, which is otherwise used for audio streaming – and it can only function in bulk data USB transfer mode, which carries no timing guarantees (and may retransmit erroneous data, which is not done when streaming). In this respect, the work in part II demonstrates that *even* when working with bulk USB transfer, at 200 kB/s bandwidth it is possible to obtain reliable audio streaming of up to and including one-quarter of CD-quality bandwidth.

Of the papers in part II, the first, paper II-A, is somewhat distinct: it is a reimplementation of an older project, for which the code is archaic enough that its easiest to compile and run it under a single-user, single-task context in an OS such as FreeDOS or MS-DOS (and as such, there is no kernel driver, even in the Linux version, running on a 2.4 single-floppy disk distribution called `nanobox` [101]); additionally, the hardware requires the long obsolete ISA bus, and features neither an on-board clock nor buffer memory (and as such, the PC software is in control of the audio sampling rate). While the software functions merely as a signal generator, in principle it is possible to modify it so audio samples are reproduced through the card – and thereby, this system can be seen as a sufficient model of a soundcard. This model

conception is important, because paper II-A proposes and demonstrates a specific hardware intervention, applicable to soundcards generally. Namely, all consumer soundcards feature so-called "coupling capacitors" on their analog inputs and outputs: they function as high-pass filters, and thus eliminate the constant, a.k.a. direct current (DC), component of the signal. This is done mainly to protect electromagnetic loudspeakers; but at the same time, it makes consumer soundcards unsuitable for generic data acquisition – given that in many sensors, useful information might be encoded in their constant, or slow changing, DC component. Paper II-A proposes use of digital switches, to allow for bypassing of the coupling capacitor filters altogether, controllable from software; and this kind of intervention would allow use of a soundcard as a generic sensor and actuator interface, which could be particularly useful for research. This perspective is the background motive for the summary of papers related to media technology research, presented in the next section.

4.2 The soundcard as a research tool in media technology

Development efforts in the area of electronic music instruments, just like those in the area of audio recording and reproduction equipment, can be considered to belong to a wider area of media technology research. All the subfields herein can be seen to share some common properties: for instance, the dependence on advances in materials science, electronics engineering and computing science; which can be seen as a base for applied research in media technology. The papers in part III can be recognized as a small sample of the volume of work in this wider research field, where the soundcard is either implicitly used (i.e. it is used a commodity tool, while not being the focus of the research), or where soundcard technology is applicable. In part III generally, papers III-I and III-J discuss electronic music interfaces, while papers III-K, III-L and III-M are concerned with the use of audio and haptics as an immersion factor in virtual reality technology. The work in part III is briefly summarized in the following text.

Paper III-H is slightly different from the others, and it is a sort of a precursor to the tutorial aspect of papers in part II. It explores wireless multi-channel sensor connectivity to the commercial application Max/MSP, running in a Windows XP OS. The problem is addressed by submitting a pair of PCB designs, one performing the role of sampling and transmission of data from sensors; and the other as receiver of data and converter to the RS-232 serial format, that ultimately interfaces to the PC application. The boards utilize a pair of frequency modulation (FM) transmitter/receiver modules to establish a wireless link, through which RS-232 formatted serial data is transmitted. The data formatting is performed on the transmitter board, with software running on a Microchip PIC microcontroller; the receiver board merely performs electronic signal scaling of the received data (using a signal level converter IC), so it can

be interfaced to a RS-232 serial port on a PC. While the educational intention is present in this work, it is hardly useful from an open-source perspective (even though there is a focus on freeware approaches): schematics and PCB layouts (which include a hardware programmer for the PIC microcontroller, usable through a parallel port of a PC) are shared[1] merely as bitmap images, not in source file format; Max/MSP patches are provided, and so is the code for the microcontroller written in the BASIC language - but its instructions refer to use of freeware (and non-open source) tools for its compiling and "burning". Open-source compatibility (and the fact that the word "driver" is used in a mechanical sense, not in the sense of a specific hardware-related program running in the context of an OS kernel) aside, the project described in paper III-H advertises an 8-channel, 8 bit sampling rate of only about 11 Hz (limited by the serial link speed of 9600 baud and inefficient data encoding). An availability of open soundcard projects at the time, that could have served as a inspiration base or precursor to this project, might have resulted with more informed development - and devices with far superior sampling parameters, that might have been usable in research as tools, beyond the role of a mere exercise.

Paper III-I belongs to the domain of electronic music instrument research; in particular, it summarizes results of two short-term research visits, where audio physical models were driven by DJ gestural control. The experiments in this study were performed using Skipproof, a virtual turntable DJ application implemented in Pure Data; a prototype of a Reactable digital instrument, also with a software portion running on a PC; and a code implementation of a friction-based model of a violin. In all cases, a soundcard was implicitly used to generate the audio playback output. While the Reactable prototype is a tactile interface in its own right, but one where data is acquired by a video camera (and thus with an effective sampling rate limited by the camera frame rate and video processing speed) – the Skipproof program is controlled mainly through a GUI using a mouse; which affords possibly a greater sensor sampling rate, but also a reduced convenience in terms of musical tactile interaction. This limits the sampling rate of the sensor signals for interaction input at around 25 Hz (and up to the order of some 100 Hz) for a standard video camera based input; while mouse signals are delivered within Pure Data with a minimum period of 1 ms, which results with a 1 kHz effective sampling rate. A pre-existing open soundcard platform with better sampling characteristics would have been difficult to integrate with the Reactable prototype's video based input processing. However, Skipproof has been used with sensor-equipped turntable controllers, where it is possible that a soundcard based data acquisition could have delivered input to Pure Data in the audio domain, thus potentially at CD quality - which would have improved the quality of DJ gesture captures used in this application.

Paper III-J also deals with electronic music instrument research, in the same context as paper III-I: it is a report on a short-term research visit; and discusses

[1] http://smilen.net/wdaq/

the development of Reactable objects specifically intended for a violin physical audio model (of which those related to DJ gestures are only a subset). An overview of an expert user evaluation is also given in this paper. Since the technology used in this inquiry is the same as the one used in the previous paper, the applicability of an open soundcard platform in this scope is likewise the same.

Paper III-K is a description of a system, which in subsequent papers is used in the context of virtual reality research. The system is intended to simulate the walking experience on different surfaces, and consists of a pair of shoes equipped with sensors (a pair of FSRs for each shoe) and actuators (a pair of electromagnetic recoil-type actuators for each shoe), connected to PC software implemented in Max/MSP as a development environment. The shoes allow that a real-time signal can be acquired from the walking action of a user that wears them, which is then processed by software; and upon predefined conditions - when the user makes a step motion with the toe or the heel - appropriate audio signal is generated that drives the actuators, which would thus simulate the impact between the shoe and the surface of the floor as a tactile haptic feedback. Therefore, the use of a soundcard is explicit - here, a high-end, 8-channel Fireface 800 soundcard is used to generate 4 individual audio channels at CD quality, that drive the total of four actuators in the shoes. On the input side, the FSRs essentially measure the force applied to either a toe or a heel region of a shoe, and thus provide a signal from which it can be deduced whether a step has been made. However, the FSRs provide the measurement of the applied force as a DC voltage level - which would get filtered/distorted by the input coupling capacitors of the soundcard, if an attempt was made to use it to additionally sample these sensor signals. Therefore, in spite of the availability of a high-end, multi-channel soundcard, a secondary system is used here for sensor acquisition, consisting of an Arduino Diecimila board - which can capture the unfiltered DC levels of a signal, however at inferior sampling parameters than the soundcard; and deliver them to a PC via USB/serial connection. Clearly, an existence of an open soundcard platform - especially one where the input coupling capacitors can be bypassed at will from software, as paper II-A suggests - would be particularly applicable to this kind of a device: as it would have allowed both multi-channel sensor sampling, and actuation, from a single platform, which would have dealt with all signals as CD quality audio signals. From a perspective of a researcher, this might have reduced the software implementation complexity as well (e.g. as all signals would be audio signals, work specifically on interleaving serial and audio data in software would be unnecessary).

Paper III-L describes a preliminary experiment on audiotactile cues in virtual environments using the system described in the previous, paper III-K. A group of 45 volunteer users - wearing the augmented shoes, and headphones for audio feedback - were tested on their ability to discriminate between different floor surfaces (such as wood, metal etc.) reproduced through the system, while passively sitting in a chair. Within this experiment, the coupling between the

haptic and the audio feedback was also explored. As the system used here is the same one as described in the previous paper, the same conclusions regarding the possible use of an open soundcard would hold.

Paper III-M is similar to the previous, paper III-L, in that it describes an experiment with the augmented shoes system, described in paper III-K. The difference is that this time there are 30 participants in total, and they are asked to walk during the evaluation procedure. Thus, the audio (and haptics) synthesis engine has to perform in real-time, in response to the sensor signals obtained from the augmented shoes. The laboratory walking area was limited to some 18 m^2, and the wires necessary for input and output for each shoe were collected in cables up to 5 m long, connected through DB9 connectors; this does present a degree of hindrance to the free walking of test participants. Also here, the coupling between the haptic and the audio feedback was explored; and the participants were asked to identify the material of the floor surface they experienced to have been reproduced by the system. Since the system used is the same as in paper III-K, the same technical conclusions about applicability of a possible open soundcard platform would apply. However, since here walking is in focus, and cables mounted on shoes do represent a hindrance to unrestricted walking, also the approach of paper III-H is applicable, in the sense that wireless data acquisition would have provided for a freer walking experience during testing (and this was indeed attempted later in the laboratory, among other things by using so-called wireless "shields" for the Arduino board, such as the Watterott RedFly.

This overview of work in part III outlined efforts with diverse research foci within media technology, however all closely coupled with both sound, and use of sensors for interaction. As such, the potential benefits from an existence of an open soundcard platform – one with a level of hardware and software design openness of the kind promoted by the papers in part II, but with a reliable CD quality operation; allowing hardware implementations of algorithms as described in chapter 3, and modifications like e.g. the bypass of input capacitors as in paper II-A or extension with wireless operation as in paper III-H; but above all with a clear scalability plan, with known limitations to extending the platform with additional number of channels in terms of both hardware and software – have also been identified. The fact that such a "dream" platform was not available - and not even identifiable - at the start of this PhD project, was one of the main motivators of the research conducted in part II. This thesis, unfortunately, cannot claim that it brings about an open soundcard platform sufficiently embodying these specifications - however, it is implied that a study of the kind as in part II in a *necessary* step towards such a platform, should it ever emerge. The emergence of such a platform may, in fact, be determined by factors outside of the scope of user requirements and technical analysis - some of these are briefly touched upon in the next subsection.

4.3 Open development perspectives

The papers, summarized in the previous sections, illustrated the open approach to the soundcard – in the role both of a subject of research, and as a potential tool in broader media technology research. As a subject of research, an effort is made to consider the soundcard as a reliably reproducible laboratory exercise: first, by considering the soundcard as a generic device, and further, by identifying actual hardware and software systems that demonstrate soundcard operation. This assumes that the development of technology has progressed to a point, where means to reconstruct operation of hi-fi soundcards which first emerged commercially in the 1990s, are available to an average hobbyist today; if such means are identified, they can thus be advertised as a basis for laboratory exercises in educational institutions. Thus a degree of openness is already predicated upon research of this character: without full disclosure of hardware and software details, it would be difficult for others to reconstruct the exercises – and it would be difficult to exercise the freedom to modify the designs to suit one's needs.

One of the most apparent limitations in an undertaking of this kind is economic cost. In this context, "hobbyist means" would describe, as a rule of thumb, an allowance of about €30 to €50 a month, with occasional purchases €100 and above (which start triggering the pain threshold); this, along with an access to electronics shopping opportunities, could allow for amassing the equivalent of a personal low-cost lab bench (tools like soldering iron or multimeter and perishables like PCBs or soldering tin) over a course of a year or more through unrelated, smaller projects. Additionally, ownership of a PC, as well as paid access to appropriate space, the electric grid and Internet is assumed; thus "hobbyist means" could probably be within the financial reach of the majority of the individuals in the developed world. Similarly, institutions are assumed to already possess lab facilities of equivalent character, available for student use. Clearly, this kind of a dependency represents an a priori expense; and a conduction of the soundcard exercises can only increase the cost. If limiting the expenses is the only criteria, then by using free & open-source software which can be legally obtained and used essentially gratis (more precisely, for the cost of Internet traffic); and hardware, like the Arduino board or the Xilinx FPGA chip, that individually lie within the hobbyist means – the papers in part II could already be declared a success. That, however, must be seen in the context of current technological progress – where more aspects of free & open-source approaches become relevant (and not only to financial cost), than just the possibility for gratis software downloads.

Note first that from a lab exercise perspective, where generic concepts would be demonstrated, there is not much use in claiming originality or novelty; as such the development process would benefit from maximum reuse: there is no need to waste time on developing a PCB design, or an programming an entire operating system, if such designs are already available. Let's recognize that the extent to which this is *possible*, is influenced by the intertwining of

technological and legal developments, - and the resulting change in social outlook and business climate, - historically originating mostly from the United States of America. Additionally, in this specific context, there is an implicit chicken-and-egg conundrum: computing hardware, even if powered, on its own is useless without a software program specifying what it should do; a source code of a program is likewise useless without computer hardware to run on, or without tools to compile the source code into executable machine instructions appropriate for the given computer hardware architecture – and this extends to any subsystems that may be integrated as a part of a computer, such as the soundcard. Therefore, software code originally *was* open-source: in the past, computer hardware was so big and expensive, only institutional customers could afford it; as this also resulted with a scarce amount of programmers, manufacturers were forced to bundle source code of programs along with the hardware – lest the customers find the big, expensive, hardware useless.

All of this changed in 1969: this year, anxious of the significant market share bordering on monopolization, the U.S. government raised an antitrust lawsuit against the IBM company; in response, IBM decided to unbundle software product sales from hardware [102] – this event is believed to represent a crucial point in the growth of the business software market. Thus software, that hitherto had been perceived as gratis by customers, now incurred a monetary cost. The effects of this, however, were not instant - allowing a culture of hacker ethics to develop throughout the 1970s [103] among research departments and hobbyist communities alike, where software code was freely exchanged and modified. Meanwhile, businesses started realizing that by not providing the source code, but instead only the executable format of software, they both spared the customer of the time and labor involved with compiling a program - and increased the dependency of the customer on support from the company, which is a clear opportunity to increase the extortion of profits on an already made sale – while simultaneously making it more difficult for the competition to merely copy that software in competing products and profit from it. Imaginably, the contact between these two ideologies was not a happy one; for instance, in 1976, Bill Gates, the co-founder of Microsoft, published the "Open Letter to Hobbyists", accusing the hobbyists of thievery due to their copying of the BASIC interpreter software for the Altair computer [103]. Another oft-cited episode is the arrival of a Xerox printer as a gift at the MIT Artificial Intelligence Lab around 1977; whereas the previous model driven by open-source software allowed the researchers to add relevant functionality, the new model using closed, proprietary software disallowed that - even if in principle the researchers were perfectly capable of implementing the functionality, if provided with the source code. Unsurprisingly by today's standards, the Xerox company was entirely uninterested in doing any modifications on the behalf of the researchers as users.

It is episodes like these, which exemplify the frustration of being *capable* of implementing a relatively straightforward modification on a technology, *but unable* due to unavailability of a sufficient basis - not because of natural, but

because of strategic and legal limitations - may have been the motivating factor behind the emergence of the Free Software Foundation and the corresponding movement and philosophy, described in more detail by its founder Stallman (2010, [104]). Similarly, the frustration due to lack of a soundcard platform, open enough to allow for generic understanding of hardware that implements real-time algorithms as in chapter 3, or for hardware modifications as in paper II-A, can be seen as the motivating factor for the open soundcard development carried out in part II. This frustration can be said to be borne out of a desire for efficiency - at least in the sense of not having to duplicate already existing work, that is, not having to reinvent the wheel; this possibility is in a sense native to software, and as a development approach can be called, as per [104], "standing on the shoulders of giants". This is clearly seen in the context of reuse of software: if one has access to the source code, one can modify the software instead of rewriting it from scratch, and thus avoid waste of time and labor; there are nuances, however, in how this can be addressed.

In a sense, these issues can be reduced to the question of who owns the control of a product of labor in a producer-consumer relationship. In other words, it is the question of who is a thief, or worse, a freeloader or a parasite - a term of endearment spanning both left-wing (e.g. “the rentier state is a state of parasitic, decaying capitalism [...]” [105]) and right-wing (e.g. “[...] if they don’t want to take menial or low paying jobs; arrogant parasites like that should starve to death [...]” [106]) political and economic views. From a more menial perspective, these relations may be easier to understand: if I as a peasant plant, tend and harvest my own food, it is mine by virtue of my labor on it: I should be able to consume it any time I want; if I as a blacksmith mine my own ore, smelt it, and forge it into a hammer head, and chop my own wood fashioning a handle, assembling the whole as a hammer, it is likewise mine by virtue of my labor: I should be able to use it anytime I want.

However, while nature may seem sympathetic to life in general locally (after all, we’re here), it also seems hostile to individuals - not the least, due to the requirement of nutrition for sustenance, and the corresponding existence of a food chain. Additionally, there is only a limited window of opportunity to consume food before it spoils, and once it is consumed, it’s gone - for continued life sustenance, fresh food material must be repeatedly obtained and consumed. Thus, I as a peasant might be called self-sufficient (as long as there are conditions that allow my labor to result with continuous resupply of food) – but I as a blacksmith am not, considering that the time required for the required job operations will preclude me from simultaneously laboring on a garden. Because of this, I as a blacksmith am forced to depend on the labor of the peasant, and I might as well trade my hammer for some quantity of carrots. But, I wouldn’t be able to survive on carrots alone, forcing me to trade with other peasants, too; and it doesn’t take long for the idea to abstract the value of a product into a token, - whether shells, precious metal nuggets, paper money, or bits of data (stored possibly as a magnetic pattern on a disk), - that can be used to trade for products. The problem is that as an abstraction, there is no a priori physical

determination of what is the value the token represents; in reality, its value is negotiated between the participants in the economy of the given token - or in other words, all money is fiat (which in recent thought, starts getting addressed by applying the oil industry's concept of energy return on energy invested (EROEI) to the notion of currency, e.g. [107]). And once there is an environment where trade is viable (meaning that there is already some excess of production that can be traded), it doesn't take long for one to get the idea that one's labor can be avoided, if those laboring are compelled to part with their produce as a tribute; for instance by - in the best racketeering tradition - offering protection to the peasants from harm; harm which, in lack of external enemies, may just as well be inflicted by oneself to begin with. No more than an image of the ancient Spartan institution of κρυπτεία (Krypteia, see e.g. [108]) - where youthful secret officers terrorize a peasant Helot population as a rite of passage - needs to be conjured, to recognize that the origins of civilization are rooted in the monopoly of an élite on exerting violent force on a mass of peasants; from a Machiavellian perspective, one's feelings on the matter wouldn't matter: might de facto *is* right, and thus cruelty is apparently a virtue.

Especially as the civilized state comes to accept - and indeed, demand - its tributes paid in currency as tax, it is not difficult to imagine how most individuals in this social context would naturally come to perceive money itself as value, rather than as as a mere token of value. With that, come the inevitable attempts to hoard or manipulate currency, possibly causing an even greater disconnect between the perceived and actual (if such can be defined, provided it is an abstraction to begin with) value of currency - resulting occasionally with catastrophic inflation, examples of which are known for millenia (at least since the Roman Emperor Diocletian's coin debasement [109]). This disconnect can become exacerbated when technology is introduced, where assembly-line industrial processes can bring about previously unachievable propositions: if a manually copied book used to cost me 100 tokens, and a newly printed one costs me 29 tokens, it clearly costs me less as a consumer – but does that correspond the actual overall cost, if the existence of the printed book product is predicated on an industrial level investment in a print shop, say of a million tokens?

And this disconnect is possibly most obvious in the production of items that are the physical carriers of information directly decodable by humans, in modern terms - *media*; and the notion of copying. In ancient terms, if I have already procured parchment, quills and ink (all of which have a "hard cost", as they need to be processed from animal carcasses), I borrow a book from you, and I proceed with using my own labor to copy its contents manually, returning the book to you once I'm done - I may have inconvenienced you a bit, but I'm hardly a long-term parasite; in turn, I should own my copy of the book, by the virtue of investing my labor and resources in it. In that regard, if someone else permanently takes away my copy of the book without consent, so I cannot read it any time I please, it would represent theft from me - not from you as the owner of the source of the copy, which would still remain in your possession regardless. Or at least, it might be a reasonable position to assume

today; Irish myth preserves the story of the battle of Cúl Dreimhne [110] around 560 CE, which is fought in response to a ruling that proclaimed: "To each cow its calf; to each book its copy". Note that the concept of copying cannot be applied to other types of physical products; one needs to invoke the notion of a molecular assembler (e.g. [111]) – but one that would allow me to put in any cheap rock as source material, and get an edible "copy" of an apple owned by you as output; so that both you keep your original, and I also own my copy (in full analogy to the concept of copying information in media) – something that might bring about an economy that could be described as "post-scarcity"; but clearly something that is currently also a science-fictional [112] proposal, still.

Now, consider what happens when a technology allowing mass reproduction and economy of scale, starting from the Gutenberg press (and onwards to assembly line presses), is introduced to the concept of book copying: suddenly, volumes that might have required thousands of manual laborers, can now be produced by possibly tens of employees in the same time span - but only if the investment is made in the building of a press to begin with. Surely, the print shop stands to profit from this - at least as long as the consumers still have an active memory of the comparatively higher cost of the previous technology, and before diminishing returns take their course; and indeed, historically they did - "... without the Consent of the Authors or Proprietors of such Books and Writings, to their very great Detriment, and too often to the Ruin of them and their Families ...", as one of the first legal copyright acts in Great Britain, the Statute of Anne from 1710, notes. As such, with the emergence of copyright, the author becomes vested with a power inapplicable to, say, producers of agricultural produce: the temporary exclusive right to collect royalties on *copies* of a product. Note that in this period, copyright is considered a time-limited right granted by the state, for the benefit of society, as the wording "An Act for the Encouragement of Learning..." of the Statute of Anne, or "To promote the Progress of Science and useful Arts ..." of the Copyright Clause of the U.S. Constitution (1787), indicates.

With the development of industry, technology and law, this concept has evolved to the notion of *intellectual property*, resting essentially on three pillars: copyright (where a unique expression of an idea, but not the idea itself, is governed); trademark (where a unique identification of an entity's product or service is governed) and patent (where an novel idea, or an improvement of an existing one, is governed) – and additionally, has extended beyond the domain of literature into music, film, industrial (e.g. electronic circuit) design, and computer software. All these legal concepts might otherwise indicate that the author is indiscriminately privileged (at least temporarily), but that is not exactly the case in reality: since ownership of intellectual property titles is considered property, it can be legally traded - bought and sold; and as such, it is not the author which is irrevocably privileged, but the *owner* of the title. Even leaving aside the fact that in the concept of "work for hire", it is not the author of the work, but the employer paying for it, that owns e.g. the title to copyright - a lot of individual authors, especially in the domain of artistic expression, have

no other option but to cede their copyrights to a publisher in hopes of achieving success; from this point on, the publisher legally owns the copyright and can profit from it, which is not synonymous with corresponding profit of the author - for examples from the popular music industry, consider Love (2000, [113]). Eventually, this results with an ecosystem of claimants to financial royalties feeding from the actual purchases made, which is hard not to observe generally both as social Darwinist in nature, and as susceptible to inflation.

Ultimately, the mass market penetration of digitalization of information and its network exchange, has made a shift in perception of intellectual property law as well: while, arguably, intended originally to regulate the relations between (industrial) producers of media – in recent times, consumer users have increasing means to be producers of media as well, at least in terms of high-fidelity copying. The key issue is that with the development of this technology, the cost of making an identical copy for a typical consumer becomes vanishingly small (though never zero). This makes the boundary between the consumers and producers of media ever more blurred, thereby turning the consumers into occasional targets of copyright law: e.g., even if I alone invested in a computer, CD-burner hardware and software, and paid legally for an original CD – should I produce a copy with the intent to share, even without any expectation of financial profit, I am liable for prosecution; consider [114] for an overview of recent cases, some involving prison, or as in a case of a Minnesota mother, fines to the tune of millions of US$. If I have saved money to buy a computer - for nearly all cases, other than a custom built desktop or a Macintosh brand, I will get a Windows OS pre-installed (that is, bundled); should I ask the shop to remove the OS, if I intend to use a free alternative, I will not get a refund, effectively placing me in a position to pay a "Microsoft tax" [115]. Even if I go through with that, I would not be able to watch my own legally bought DVDs on a free and open source OS: even if I programmed such a software player myself, making software that bypasses DVD copy protection publicly accessible is enough of a reason for a court litigation [116], and usage of such existing open-source software is seen by the U.S.A. as a violation of the Digital Millennium Copyright Act (DMCA) [117].

Somewhat ironically, this could be seen as an example of the free market ultimately resulting with monopolies, that can afford to claim payment from a minority of consumers that don't want the product, but don't really have another choice either - as long as the vast majority of consumers don't otherwise complain. Complementarily, such a situation changes the perspective on ownership altogether: if I buy a smartphone, and I can never be sure whether I have turned it off or not [118] - do I really own this device? If I buy a computer with a pre-installed closed source OS, which I cannot change without reverse engineering efforts, but which "phones home" letting the producer know of every program I intend to run [119] - is this computer really mine? If I buy a song or a film in digital format, and I'm prevented of transferring it from the computer to the smartphone and vice-versa – both platforms that contain both hardware and software for reproduction of such files, but otherwise incompati-

ble between each other – am I really the owner of this copy? These concerns may be more than just a cynic's rhetoric; recent developments indicate that copyright ownership ideals seem to be spreading to less abstract industries – for instance in the automotive industry, "GM owns the copyright on that code and that software ... a modern car cannot run without that software; ... therefore, the purchase or use of that car is a licensing agreement" [120].

It is this kind of intertwining of the art and technology industries, that - beyond companies that patent trivialities like rounded corners, and sue each other over that to the tune of billions of US$ [121]; and companies that as non-practicing entities (a.k.a "patent trolls") exist solely to profit through patent litigation [122] - ultimately results with restrictions for the user, which the free & open source software approaches attempt to partially address. Note that "free & open-source" is a sort of a blanket term, which may encompass diverse approaches: the licenses involved could be seen to form a spectrum (see e.g. [123]), containing several distinct points:

- A "public domain" license imposes no restrictions; anyone is free to modify the work and restrict others' access to the modification by covering it with a proprietary copyright;
- Free software, exemplified by the GPL (General Public License) and related licenses (promoted in e.g. Stallman (2010, [104])), emphasizes the freedom of the users (anyone) to copy and modify the software, by imposing the restriction that all modifications to the work must be released in like, free manner;
- Open source software, exemplified by BSD and related licenses (promoted in e.g. Raymond (2001, [124])), emphasizes the freedom of developers to relicense modifications under more restrictive, proprietary licenses, as long as the original work is credited.

In other words, even if the deliverables related to the papers in part II have been released under the GPL simply as credit to the license, under which the vast majority of the operating system and tools used have been originally released - the choice of a given license carries specific legal implications; which in this case, however, happens to align with the educational intent of exercises described therein. Under all of these licenses, anyone is allowed to package the work and sell it for financial profit; at the same time, the licenses do not restrict anyone else to package the work and share it with peers, which on the Internet can be conducted nearly gratis – which, along with the requirement for license attribution in free & open-source software, may severely limit the potential for profit on sales of copies of the work alone in this context (similar but not identical to the experience the music and other publishing industries with illegal copying).

This turns into one of the major points of contention when discussing free & open-source approaches in terms of economy: if the author cannot profit

from sales of copies of a work, how could even the basic expenses, such as food and lodging, going into the production of a work be covered? A typical response in this debate revolves around suggestions of alternatives, which at least in the domain of music and software are straightforward: rather than relying on profits from sales of copies, a music author might mainly subside on concerts/live performances, while a software author might subside on selling support for software – although, this approach is not so clear elsewhere, e.g. in traditional literature (while selling hard paper copies is undisputed, offering the same content as a gratis digital download in parallel might clearly harm sales). However, recall that even before digital information technology disrupted the traditional hard copy publishing, it was a business with very few guarantees for authors: there may have been few top sellers like the Beatles or the Rolling Stones bands, but they are certainly outliers; there may have been more that managed to make a living continuously from authorship royalties, but hardly in the numbers to make it a stable profession; and what remains is essentially a sea of commercial failure. And even when discussing commercial successes in this scope, recall that it is generally the owner of the copyright that profits, for instance the publishing company - not necessarily the authors. Thus, any implied guarantees for financial profit just on the basis of copyright, from an authors perspective should be seen as illusion; instead of a guarantee, success for a particular author may just as well be a question of statistical chance - or in plain terms, luck.

At least in the context of this project, however, the issue of economics might be more straight-forward: the deliverables from part II can be perceived as tools, developed in the course of research, funded by a state-run university - and thus ultimately by the tax payers. As such, the development of the software has *already* been paid for; at least in principle, offering unrestrained public access to it should be one of the default positions regarding its use - although, note that a possible point of contention is that the result of free & open-source licensing is access for anyone, not necessarily just the tax payers that actually paid for it. However, the author decision to open-source material should be individual and conscious, a matter of personal freedom; enforcing open-sourcing as a large-scale policy, not the least in academic environments, would be under risk of severely negative reactions. Otherwise, should the legal privilege for a restrictive copyright be preferred, the employer would clearly have to be considered as claimant to a share in the profits, the arrangement of which might have to involve legal counsel. In this project, the decision to open-source was possibly made easier by the fact that there is nothing to sell *per se*: the performing of the exercises is predicated on already made third-party purchases, and the generic nature of the demonstrations leaves little room for profitable market differentiation, whether in terms of hardware or in software.

On one hand, the free & open-source development model is decentralized; on the other hand, developers typically focus on free alternatives to, not necessarily free clones of, proprietary software. Since anyone can freely propose and contribute software in this ecosystem, this results with a great diversity

of tools, not always compatible between each other - which is often known as fragmentation. Thus, for anyone transitioning from a proprietary to a free OS, there is a cost involved: there is a cognitive strain related to both the choice of particular OS components, and the steep learning curve, which ultimately results with at least a cost in time. Furthermore, there is the issue of having to learn to live with software bugs. But in this respect, there is after all not much difference from proprietary OSs: there is cognitive strain involved in choosing and learning tools on proprietary OSs as well, and both proprietary and open development teams are likely to ignore or refuse to fix bugs. From a user perspective, then, what matters in this: with proprietary systems, above all this, there is both a financial expense, and no recourse for the user since the source code is closed and thus unavailable; in free & open OSs, the financial expense can be reduced to essentially gratis - and if a bug is otherwise unfixable, the users at least have the option to learn enough programming to fix the bug themselves, given that the source code is available. Note that the free software approach is, in this respect, predicated on post-scarcity (which is acknowledged in Stallman (2010, [104])): it assumes that computers (and electric power) are abundant and cheap enough, that anyone could afford to learn programming languages, in sufficient degree to maintain their OSs - and should this be expected on a mass scale, comparable to native human languages, then the issues in free software can be reduced to a right to read [104], and the right to free speech.

Regardless, it is difficult for average consumers to transition to free operating systems, judging by the relatively insignificant market share they have on desktop PCs. Beyond the general reasons above, there historically was another critical reason for low adoption: operating system driver software, that enables a PC to operate with external/add-on input-output (I/O) hardware boards. In fact, "Linux does not have drivers" has long been an argument against adoption, and even as late as 2010, drivers have been considered a serious issue [125]. For the market leader, the Windows OSs, Microsoft typically outsources the driver development to the original equipment manufacturers (OEMs) of the add-on hardware; which, in turn, may outsource the driver development task to offshore developers [126]. In this context, OEMs may not find the additional expense to develop free OS drivers worthy, provided they already invested in development for drivers for the dominant market leader; this can be seen as a feedback loop that results with even less drivers for Linux - often forcing Linux developers to reverse-engineer the Windows drivers [127]. In fact, this project did release open-source software called `attenload` [128], which in order to allow a GNU/Linux software to fetch data from oscilloscopes of the brand Atten (and possibly others), required reverse engineering of the Windows USB driver for this device, and reimplementing it as (user-space) USB driver in Linux; this experience confirms that reverse engineering can be, and is, a rather significant waste of time - and ultimately, a wasteful duplication of efforts.

The exercises in part II address precisely this problem area, at least in the domain of audio; and while from an engineering theoretical perspective, the

novelty factor is nothing special – consider that the work in part II, took around 7 years to compile; but as defended in paper II-F, this work should be distilled enough to be reasonably presented in at least a week of intensive exercises, and at most several months at a leisurely pace. Allowing students, whether hobbyist or academic, to achieve a basic, yet practical understanding of the interplay - between user-space software, OS kernel driver software, and actual hardware for digital audio - in months instead of years (which would be likely for anyone attempting to do the same "from scratch", i.e. from a point of ignorance about particularities in open source development), is the primary contribution intended from this part of the project. Covering both software and hardware issues, this work is slightly slanted towards electronics - meaning that it assumes a somewhat stronger background in electronics, while taking a position of greater ignorance in terms of computer science. While related documentation certainly existed previously, it often suffered from the chicken-and-egg problem (e.g. the ALSA driver documentation explained software concepts in terms of hypothetical hardware; other resources might use actual, but undisclosed, hardware), and as such was difficult to use as introductory material; the possibly original contribution of the work in part II is the identification of *both* software *and* hardware that illustrate a complete soundcard operation, followed by a level of discussion in the papers, that is hopefully more accessible to a novice student. But ultimately, the work in part II is in itself a proof that already in the period 2007-2010, there existed open-source technologies - including a relatively stable OS that can be used for development (in other words, an OS that can be downloaded, and offers development tools such as text editors and compilation software nearly out of the box, that allow a relatively bug-free development user experience) - that could be used for technical exercises and demonstrations in streaming digital audio. Thus, the user - the potential student, or lab administrator - besides the hardware considerations, could be expected to, simply: download an official, "vanilla" free & open-source operating system; run it on a PC through a "live CD" or "live USB", which would leave any previous installs of possibly proprietary OSs on the PC unchanged; and install likewise free development tools in the OS; in order to do the exercises in part II.

Consider that one of the most popular proprietary OSs, Windows XP, reached its end of sales in 2008, and end of extended support - which marks its end of life (EOL) - in 2014. This essentially means that the users of this OS, after this date, are left to their own devices; given that malicious software (like viruses) that exploits vulnerabilities in the OS is developed constantly, and the source code is closed, there is in fact little that users can do but eventually be pressured into upgrading – also because software developers, in particular antivirus and hardware driver vendors, can be expected to phase out support for this OS as time passes. There could be an economical incentive for a monopolist to embrace this strategy of planned obsolescence, beyond just technical reasons [129] - however, in the case of Windows, upgrading the OS may also enforce hardware upgrades, which on one hand increases the cost of upgrades, and on the

other hand also pushes old computing hardware into obsolescence, possibly with an ecological impact [130]. One could surmise, that by staying generic and open, this project would spare the users (including education institutions) from both excessive legal considerations, and costs of upgrade due to planned obsolescence; and would remain technologically relevant, at least in the narrow scope of illustrating soundcard operation of up to CD quality. That is, however, not entirely true – even in the context of free & open-source development.

From a legal perspective, it is noted by Stallman (2010, [104]) that free software approaches are a "hack" of copyright law as permitted by the U.S. Constitution (a "hack" since a free license is used to promote free copying, whereas copyright otherwise is intended to do the opposite: to suppress copying). As such, this is something enforceable primarily in the U.S.A.; while other countries may have related legal mechanisms, these can be expected to be most similar in the sphere of influence of the U.S.A. (or more broadly, the Anglo-American hegemony), so in general these mechanisms are not guaranteed to be tolerant of free & open-source approaches. Ultimately, however, not even the subjects of the U.S.A. can be certain that the tolerance will continue indefinitely: consider that in the case of Wickard v. Filburn from 1942, the U.S. Supreme Court decided that a farmer, by growing additional wheat, harmed interstate commerce - even if that wheat was intended for personal use, and not for sale (and thus would never have entered the market anyway). It doesn't take much fantasy to imagine the same argument applied to free software, which encourages copying and may therefore harm commerce - and if the U.S.A. itself is susceptible to this argument, it is hard to imagine the other world powers being any more lenient. But beyond this currently hypothetical threat, there are more immediate threats. Namely, one could, in all honesty, perform one's own reverse engineering and write one's own code for a program (e.g. a driver) without external consultation: while this code would represent a unique expression entitling one to a copyright - it may still *simultaneously* infringe on a software patent. This might make distribution of such software illegal, unless royalties are paid to the patent owner; this is however more of a danger to big companies, given that patent trolls would likely wait before initiating litigation (both to accumulate evidence, and to be sure that there is actual profit to be made), which may cause companies to form patent pools [100]. Additionally, the mentioned case for DVD playback on GNU/Linux may be illegal beyond patent issues - as it involves a breach of a legal state act (the DMCA). However, in general, the current legal attitude does seem to be tolerant enough of free & open-source software, especially taking the success of Google's Android OS for smartphones into account, which uses Linux as its kernel (even if this success isn't without challenges [131]); as well as the fact that for instance in the EU, software as such is, so far, not patentable.

However, for a project of this kind, it is maybe more important to recognize that obsolescence is also a major factor in free & open-source development - even if not motivated exclusively by profit. In fact, in just the 7 years the project took to complete, a major share of the technologies used were made obsolete,

even if at the start they were contemporary:

- The used Arduino Duemilanove board is not produced anymore; the driver in e.g. paper II-C is written in respect to the FTDI USB chip on this board, which is not present on the current generation of boards (e.g. Arduino Uno). As such, the driver is not compatible with current Arduino boards
- The Linux kernel made the transition to version series 3.x; there are changes in the driver programming model, which makes the drivers released via e.g. paper II-B or II-C, written for series 2.6.x, not compile "out of the box" on contemporary kernels
- The Python programming language made the transition from version 2.7.x to 3.x; there are sufficient changes in the language, so that some scripts released along with papers in part II, written exclusively in 2.7.x syntax, will fail if ran by a 3.x interpreter without a rewrite
- The Gnome desktop environment made the transition from version 2.x to 3.x; besides setting additional requirements on video hardware, there are also sufficient changes in the API - so that, say, Python scripts that exclusively use its 2.x toolkit to provide a GUI, will not work on a 3.x environment

These are the most relevant developments in terms of the work in part II, however, there are more - often bringing contention in the respective communities to the point of a split: e.g. as a sign of dissatisfaction with the direction of Gnome 3 development, the Gnome 2 codebase was forked as the foundation for the MATE desktop project; the decision of major distributions like Debian to adopt `systemd` as the user-space startup process instead of the traditional `init` daemon, led to a fork of the Debian distribution called Devuan; not being satisfied with the development of Wayland (the replacement of X Window System, a crucial graphics component in open source OSs), Canonical (the company packaging Ubuntu) developers started developing the alternative Mir, which had caused friction with Intel developers. In a sense, this fragmentation into incompatible diversity is maybe the cost of freedom - or alternatively, progress: thus, in spite of intentions to the contrary, the work in part II will not run on contemporary free & open-source systems out of the box; and labor would be required from any potential user wanting to reconstruct the exercises on modern systems - labor to port or translate code to ever-changing API specifications; or in other words, software maintenance.

With this is mind, a reasonable overall recommendation might be to use the specific versions of the OS (Ubuntu 10.04 Lucid and 11.04 Natty), most conveniently through booting into a "live" CD or USB flash drive; however, not even this approach is straightforward anymore. Namely, PCs from 2010 onwards typically ship with Unified Extensible Firmware Interface (UEFI) Secure Boot [132], which checks for a bootloader signature, and refuses to boot the

system if this check fails. While this feature can be turned off in order to boot legacy systems that do not support it (such as the cited Ubuntu versions), it is likely to cause additional inconvenience to users, forcing them to take the booting process into account as well. Thus, a resolve for free & open-source development is in itself hardly future-proof, and just as (if not more) susceptible to fragmentation and obsolescence as proprietary systems can be – and moreover, to underdevelopment: "[...] it isn't in any one particular company's interest to dump a pile of their own resources into fixing even one of the problems [...] because they'd be handing Facebook and LinkedIn and Amazon a pile of free money in unspent remediation costs" [133]. In other words, every technology is generational [134] - and any individual may encounter a point, where the cost of upgrading may become excessive: in terms of cognitive strain, if not financially. This is probably one of the most fundamental things to keep in mind for anyone aiming to do work in the similar scope to part II: things always change, but not necessarily for the better for *you*. The most fundamental benefit, in this context, seems to be the freedom to use outside an official approval, and in spite of the fact that websites disappear: even when Ubuntu stops offering these versions of their operating systems as official downloads, anyone could legally upload their copy of the OS images on, say, a sharing service; the code related to part II is primarily released through the SourceForge website - and similarly, even if the company stops offering this service, anyone with a copy can upload it elsewhere. This would also allow use of older (from today's perspective) computers, for the purpose of conducting the exercises in part II.

Ultimately, one would be well advised to take heed of the GPL license, which has the words "NO WARRANTY" in large, friendly letters on the cover. Indeed, there is no warranty, *ever* - not just in respect to a particular program, but in terms of development, economy and life altogether; no warranty, but for failure and death. Consider that, in spite of the often impressive achievements of computer technology, it also brought us cryptocurrencies such as Bitcoin [135], concepts like electronic high-frequency trading [136] where only those traders with access to the fastest systems benefit, and ultimately a global financial crisis that is still unresolved even in the developed world [137]. Is this not but a more sophisticated and inflated form of financial parasitism upon the peasants? And is this not made more ironic, by the ever fewer number of actual peasants, replaced by a ever more consolidated (and of course, patented) genetically modified organism and biotech industry; and even more emphasized by increased use of automation, causing unemployment to the degree where jobs are not a guaranteed means of survival anymore? And wouldn't even this project also support these tendencies, at least in the sense of slightly promoting the further glut of STEM (science, technology, engineering, mathematics) majors? It would be hard not to interpret this from the overall perspective of that "perhaps the most pessimistic and amoral formulation in all human thought" [138] – the notion of ever increasing entropy as per the laws of thermodynamics. In terms of ecological economy, it has already been recognized that waste is thermodynamically unavoidable [139]; in more personal terms, one can always

recall the reformulation of the thermodynamics laws known as the Ginsberg Theorem [140], attributed to the poet Allen Ginsberg:

1. *You can't win.*
2. *You can't break even.*
3. *You can't even quit the game.*

Free & open-source approaches are likely to remain a contentious issue: in spite of the proliferation of free software, it continues to uneasily coexist with proprietary software - just as it did before; arguably, this is driven by the dominance of proprietary hardware on the market. Note that with the emergence of crowdfunding and popularity of startups, there are attempts at open computers, for instance the Novena and Librem 15 laptops; however, considering the capitalist life-cycle of hardware startups [141], which, for the few that succeed, would end with an initial public offering (IPO) or a different type of sale, by which the ownership of the company is changed - and thus, there is no guarantee whatsoever that this kind of technology will be in any way persistent. And after all, it is possibly not everywhere, where free & open-source approaches might be desired, either: as one example, free & open-source nuclear bombs may carry ethical issues beyond freedom of speech. Sometimes the personal context is not permissive to open-sourcing, either (see e.g. [142]). However, there is at least one criterion which might clarify where free & open source alternatives may be desired - and that is when a technology has penetrated lives of people, to the degree where it is required for daily activity. Thus, operating systems in general, along with text processors, web browsers, email clients etc. are prime - and indeed, achieved - candidates for open sourcing. In this context, consider that the proprietary Skype VOIP (voice-over-IP) application, recognized in the past as a disruptive technology [143] especially in respect to fixed-line telephony, changed hands multiple times - and since 2011 is owned by Microsoft. The operation of such software is dependent on the operation of a full-duplex soundcard at the endpoints; and thus, the open study of a soundcard systems becomes relevant not only as a backbone for applications in music and sensor technology, but also for one of the (still) most socially valued aspects of telecommunications - telephony.

Fig. 4.2: Hearkening back to simpler times, when bare rock and metal were enough to form a relevant electronic device (here a negative resistance tunnel diode; ref. [144]).

Fig. 4.3: Ancient depiction of wax tablet usage (c-ca. 500 BCE), that can easily be interpreted as remarkably precognitive of recent modern times (ref. [53])

Chapter 5

Conclusion

This thesis commenced with the search for a technique, that would integrate the live musical performance affordances of both traditional drum-machine sequencers, and traditional turntable-based DJ sets – along with an outline of the specific motivations and cultural background behind the emergence of this concept, which is otherwise potentially applicable to a wide range of modern popular music genres. It proposed the technique of faster-than-realtime double-buffered rendering of sequenced audio loops, which could allow near-simultaneous live experience – of both sequencing (composing), and DJ-scratching, of a piece of rhythmical music loop – to a musician. This technique would be implemented as a digital music instrument platform, which demands multi-channel and realtime manipulation of digital audio, especially in terms of playback and recording; the thesis defended that such a platform could be categorized as a digital audio workstation. As such, this research segment contributes to the domain of digital lutherie - that is, digital music instrument design and construction - as a distinct area of media technology research.

The viability of the double-buffering audio rendering technique for the integration of the two live performance styles, has been demonstrated through a released software prototype implementation. This prototype, while performing well under some conditions, clearly encounters limits where its computing performance degrades to a point, that would be intolerable to musicians for a high-fidelity, live performance application. The thesis defended that the proper way to address this would be to seek the implementation of the algorithms in hardware. There is a practical problem with this, however - hardware implementation demands both acquaintance with the applicability of diverse market hardware offerings to digital audio, and cross-disciplinary knowledge of the interface between computer science and electronics engineering. Didactically, the existence of practical exercises, that would illustrate the operation of digital audio in both hardware and software, would certainly represent a stepping stone towards an engineering of a dedicated DAW platform. This

thesis emphasized the suitability of the soundcard, as a generic concept, for such an exercise - not the least, because of its presence in the current ubiquity of personal computing devices.

Since no direct precursors of the kind were identified, the project resulting with the papers in part II undertook the identification, design, implementation and documentation of technologies that demonstrably behave as a soundcard of a personal computer. The thesis defended the free & open-source development approach in this context: at the least, the entire operation of the operating system has to be known, in order to debug and troubleshoot problems on a hardware level. Furthermore, manually implementable hardware designs would facilitate experimenting with incremental improvements, which could extend the concept of a soundcard into a generic sensor/actuator processor - and thereby, bring it further towards a functioning part or basis of a DAW platform as described previously. The applicability of such a platform to a wider area of media technology is illustrated through the papers submitted in part III – all of which use a soundcard or a similar system, while not focusing on it as such.

The main contribution of the work in part II, which took up the major share of the resources in the development of this thesis, is as a time-saving compendium; one which clearly identifies both hardware and software components, and reproducibly illustrates both proper operation and problems, in the reproduction and capture of digital audio from/to a PC. Thereby, the intended target user group (consisting of individual hobbyists or students, or possibly academic laboratory administrators) is offered a price-conscious study package, that could reduce the time required for practical acquaintance with high-fidelity, full-duplex and multi-channel digital audio to months or even weeks - as opposed to years, that were required for the compilation of the works in this project. This work illustrates proper soundcard operation with uncompressed digital audio, on a variety of technologies, with gradually increasing performance qualities: from from mono/8 bit/8 kHz, up to - but in the current state, not including - CD quality (stereo/16 bit/44.1 kHz); as well as the limits to proper operation in CD quality context, which emerge despite of the apparent availability of streaming bandwidth, due to the scheduling and preemptive nature of contemporary operating systems and the specifics of the I/O bus protocol. Practical acquaintance with these issues is a necessary - but possibly not sufficient - condition, for any prospective development of a system with a scope of a DAW platform.

Reducing PC-based digital audio to a set of exercises within the means of an average hobbyist, may have been an impossibly expencive endeavor - rife with proprietary technology lock-ins - at the turn of the 21st century, when digital audio was at an earlier stage of its takeover of the mass markets. In that regard, this thesis is a testament that already in the period 2007-2010, the overall technology growth in fact did bring about products, that allow implementation of exercises of this nature with a nearly free & open-source toolchain; this

process being additionally assisted by a growth of online fora through which peer-to-peer and self- support can function. However, at the same time, this thesis is also an exemplification of technology being generational: the pace of technological development was fast enough, that most of the technology outlined in part II is already obsolete - even if it wasn't at the start of the project. This also impedes academic dissemination of practical studies - as publishers' response, review and publication processing times can be slow enough, to sometimes approach the duration of the market lifetime of a given product. In that sense, the work presented can be regarded as emerging "too late"; but alternately, it can also be said to have been done too early: realistically, the consideration of the soundcard as a generic technology would only be possible within a stable understanding of what a generic personal computer is. And while that outlook might be possible, once there is a historical backlog of, say, several centuries worth of output of the computing hardware industry – it is clear that in this era, constant adaptation of one's conceptual and practical understanding to ever changing technological developments, is still required. However, it should be clear that the challenges to this updating process are not only of the physical nature (concerning e.g. the underlying mechanisms of processes), but are also influenced by strategic, economic and political decisions of key market - ultimately, human - players.

Ultimately, the work in part II does not result with a DAW platform, versatile enough to allow cheap and straightforward repurposing to implementation of the algorithms in chapter 3 in hardware, or as a generic processor applicable to research in part III. And even if it did, its continued maintenance - and relevance - in the face of ever changing technology would have been a considerable challenge; and the same holds when considering the reduced scope of soundcard exercises. However, it does provide a basis, that could facilitate easier reimplementation and decentralized sharing of soundcard exercises in newer, contemporary technology - which might, eventually, lead to an emergence of more advanced, yet still free & open-source, digital audio platforms.

5.1 Future perspectives

People have, possibly, always had the desire to immerse themselves in a world of their thoughts; as evidence, consider how easily can the art on Fig. 4.3, several millenia old, be anachronistically (and even, uncannily) interpreted from today's perspective, as a person immersing themselves in interaction with a laptop computer. Possibly, this may be one of the drivers behind the existence of culture - here seen as a transmission of learned information in diverse forms - in general. Music certainly has always represented an integral part of culture; and even beyond its intellectual scope, its social impact (e.g. through concerts) makes it a fundamental cornerstone of the "circenses" part of *panem et circenses*, a seemingly unavoidable occurrence in governing civilized societies. Within this context, musical instrument - and herein, digital lutherie - research will

clearly remain relevant. What extent will this relevance have, however, is more difficult to state: this very project was driven by what were once underground fashions, partially based around use of instruments not originally meant to be musical; while some of these styles (e.g. hip-hop) came to dominate the current popular music business, the fashions are likely to change, just as they have always done before.

Therefore, while this thesis facilitates a potential development of a DAW, that could be freely modified by the user to achieve various modes of live interaction - there is no guarantee that such a platform would ever emerge. From a wider perspective, however, it is clear that technology in general changes rapidly, and becomes more and more unavoidable in daily life. As such, it should be of interest to society that technology is in general *known*, and that it can be learned in practical terms. This implies a degree of "tinkering" - and this thesis defends, that even when this tinkering is concerned with technology that is not necessarily state of the art (when the overall market is taken into account), it can become academically relevant in the context of free & open-source development: because then it can be used as basis for laboratory exercises that can be freely exchanged and modified, wherefrom an educational value emerges.

Electronics technology also tends to become ever so smaller; modern hardware has strongly been moving away from designs that can be manually implemented cheaply (e.g. with a soldering iron on a through-hole PCB), and this may have the effect of discouraging tinkering somewhat. However, anno 2015, there are still many ways in which the basis provided by this thesis can be adapted - and this is probably the main future perspective of this work: a decentralized development process, generating soundcard or digital audio exercises, that keep pace with ever-changing new technology. For instance, the current Arduino generation (Uno) has an openly programmable USB-serial chip - this chip can be programmed to implement a USB audio class, for which generic drivers already exist for major operating systems, thus re-implementing this board as a soundcard; this will likely need to be repeated as e.g. USB version 3 starts overtaking the market - or in general, as new OS versions emerge. This kind of process, when supported by a free exchange of designs, is certainly one of the ways in which academia - as the main disseminator of knowledge - can seek to remain relevant even with the rapid developments of technology; technology, on which society is increasingly dependent. And this increased dependence of ours, should possibly be a motive enough to support efforts in demystifying technology – at least in those areas where it is, still, nearly indistinguishable from magic – to as wide of a public as possible.

5.2 Acknowledgements

There are many, many people to whom I owe a thanks for helping or inspiring various aspects of this Ph.D. project – and my life in general; while I will not be able to do justice to all, I may as well try in this section (in no particular order). Greetings to families, where applicable, are assumed - and apologies in advance to anyone I may have forgotten :)

First, maybe, the most unlikely: I owe my thanks to the responses (sometimes amazingly fast) and the tutelage, of the anonymous, unsung heroes of many a Internet fora – without whom a large portion of this project would not even have reached the current state. In particular, thanks to the communities of `stackoverflow.com`, `tex.stackexchange.com`, `unix.stackexchange.com`, `superuser.com`, and many other Q & A sites on the Stack Exchange network. Thanks to `sourceforge.net`, as the open-source publishing website through which this project's associated code was initially released (and from where, support from other projects was sometimes obtained). Additionally, many a thanks to the support of the mailing lists `alsa-devel`, `portaudio`, `audacity-devel`, `libusb-devel`, `comp.arch.FPGA`; as well as forums at: `arduino.cc`, `xilinx.com`, `latex-community.org`, `ubuntuforums.org`, `fpgarelated.com`, `fpgacentral.com` – and many, many others.

Beyond my mother and father, I owe thanks to my wider family: aunts, uncles and cousins, for their support of my studies in Denmark. Thanks go out, not only to Stefania, Rolf, Erik Granum, Lise Kofoed, Hans Jørgen Andersen and Michael Mullins - but to everyone I've had the pleasure to work with at Aalborg University Copenhagen (AAUC) [Aalborg Universitet København (AAUK)], dept. of Medialogy/Media Technology; starting with the amazingly supportive (past and present) secretaries, Ulla Høeberg Hansen, Ulla Schou Jensen, Marianne Kiær Schwaner, Gitte Kjær Christiansen, Judi Stærk Poulsen, Lisbeth Kirstine Nykjær, Lisbeth Schou Andersen, and Lene Rasmussen. Many thanks to IT (especially for coping with my demands and experiments regarding the Linux servers :)) and support personnel at AAUK, past and present: Lars Frøkjær Knudsen, Mikael 'Mikki' Krøyer, Søren Filtenborg Jensen, Michal Dobroczynski, Jesper Michael Alsted Greve, Alex Elvekjær, Sigurd Kristensen, Michael Ammekilde, Frank Petersen, Kadir Ates, Stig Møllenberg and others. Thanks also to the staff of the electronics laboratories at what used to be Ingeniørhøjskolen i København (IHK) in Ballerup (now part of DTU), among others Ivan Gyllich Stauning and Heinz-Dieter Surkau.

Thanks to peers, as well as professors/teaching staff (later colleagues), and pretty much anyone I've met during my M.Sc. and Ph.D. studies at AAU: Vincent Agerbech, Brian & Allan Engqvist Johansen, Nick Strange Thye, Þröstur Bragason, Norbert Krüger, Volker Krüger, Luis Emilio Bruni, Florian Pilz, Amalia De Goetzen, Viggo Holm Jensen, Stefano Papetti, Lars Reng, Henrik Schønau Fog, Thomas Bjørner, Iver Mølgaard Ottosen, Steven Gelineck, Niels Böttcher, Niels Christian Nilsson, Jon Ram Bruun-Pedersen, Bob L. & Carla Sturm, Olga Timčenko, Juraj Kojs, Daniel Overholt, Sofia Dahl, Shawn Trail, Dannie Michael

Korsgaard, Erik Sikström, Francesco Grani, Stefano Trento, Cumhur Erkut, Kristina Daniliauskaite, Eva Sjuve, Esthir Lemi, ... and many, many others. Not the least, thanks to the generations of undergraduate students I've had in my AAUK courses, for being generally very nice to me (aside from the occasional complaint :)). Special thanks go out to the hosts of, and everyone I met at, my two study visits, for their kind welcome; among others, at KTH in Stockholm: Roberto Bresin, Kjetil Falkenberg Hansen, Marco Fabiani, ... and at UPF in Barcelona: Sergi Jordà, Marcos Alonso, Martin Kaltenbrunner, Carles López ... Greetings to everyone I've bumped into at a conference somewhere for a chat, as well :)

Special thanks to Ingrid Skovgaard, as well as Biljana Tanurovska-Kjulavkovski (Бљаце), and to BB&S ApS: Peter Plesner, Thomas Brockmann, Christian Poulsen, Jan Bryld, Claus Jespersen, and everyone else I've met at the company, for the amazing internship projects and all the good times. A big thanks also to everyone I've met during my stay at Aarhus tekniske Skole (now rebranded Aarhus Tech), both teaching staff and peers, including but not limited to: Jonna Zeuthen Bach, Lars Gregersen, Anne Gerd Hultberg, Birgitte Dahl Hansen, Fariborz Sahaf, Helene Rodian, Julie Andersen, Louise Bro, Ditte Dickenson, Dúi Róbertsson, and many others. I feel I should also thank all I've had the pleasure to meet from the U.S.A. – in particular, everyone I've met during my year in Maine, especially my host family: Lou, Kathy and Crystal Young; and all artists I've met while doing promotions in Skopje, like Derrick May, Kevin Saunderson, Gene Farris, Das EFX ...

By this point in time, I'd probably also have to write a thanks to the entirety of Skopje and Macedonia :) But to keep it short – beyond the people directly mentioned in chapter 2, big thanks go out to the professors and peers at the Electrotechnic Faculty (ETF) in Skopje during my bachelor studies, especially those колеги (colleagues) I've ended up in study groups with; and a special thanks to all I've hung out with in any musical subculture scene through the years - especially those people I've composed, produced or promoted music with. Also, a shout out goes out to all those, that have had the misfortune of ending up as fans of some of the productions I've contributed to :)

Thank You! ∞ Ви Благодарам! ∞ Tak skal I have!

Bibliography

[1a] Smilen Dimitrov, "Extending the soundcard for use with generic DC sensors", in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2010)*, Sydney, Australia, Jun. 2010, pp. 303–308, ISSN: 2220-4792, ISBN: 978-0-646-53482-4. URL: http://imi.aau.dk/~sd/phd/index.php?title=ExtendingISASoundcard.

[2a] Smilen Dimitrov and Stefania Serafin, "Minivosc - a minimal virtual oscillator driver for ALSA (Advanced Linux Sound Architecture)", in *Proceedings of the Linux Audio Conference (LAC 2012)*, Stanford, California, USA, Apr. 2012, pp. 175–182, ISBN: 978-1-105-62546-6. URL: http://imi.aau.dk/~sd/phd/index.php?title=Minivosc.

[3a] ——, "Audio Arduino - an ALSA (Advanced Linux Sound Architecture) audio driver for FTDI-based Arduinos", in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2011)*, Oslo, Norway, May 2011, pp. 211–216, ISSN: 2220-4792, ISBN: 978-82-991841-7-5. URL: http://imi.aau.dk/~sd/phd/index.php?title=AudioArduino.

[4a] ——, "An analog I/O interface board for Audio Arduino open soundcard system", in *Proceedings of the 8th Sound and Music Computing Conference (SMC 2011)*, Padova, Italy: Padova University Press, Jul. 2011, pp. 290–297, ISBN: 978-8-897-38503-5. URL: http://imi.aau.dk/~sd/phd/index.php?title=AudioArduino-AnalogBoard.

[5a] ——, "Towards an open sound card — a bare-bones FPGA board in context of PC-based digital audio", in *Proceedings of Audio Mostly 2011 - 6th Conference on Interaction with Sound*, Coimbra, Portugal, Sep. 2011, pp. 47–54, ISBN: 978-1-4503-1081-9. DOI: 10.1145/2095667.2095674. URL: http://imi.aau.dk/~sd/phd/index.php?title=AudioBareBonesFPGA.

[6a] ——, "Open soundcard as a platform for practical, laboratory study of digital audio: a proposal", *International Journal of Innovation and Learning*, vol. 15, no. 1, pp. 1–27, Jan. 2014, ISSN: 1471-8197. DOI: 10.1504/IJIL.2014.058865.

[7a] ——, "Comparing the CD-quality, full-duplex timing behavior of a virtual (dummy), hda-intel, and FTDI-based AudioArduino soundcard drivers for Advanced Linux Sound Architecture", *Linux Journal*, 2015, Manuscript submitted/in review.

[1] Olivia Mattis and Robert Moog, "Leon Theremin; Pulling Music out of Thin Air", *Keyboard*, vol. 18, no. 2, pp. 46–54, 1992.

[2] Margaret Schedel, "Anticipating interactivity: Henry Cowell and the Rhythmicon", *Organised Sound*, vol. 7, no. 03, pp. 247–254, 2002.

[3] Marcelo Mortensen Wanderley, "Gestural control of music", in *International Workshop Human Supervision and Control in Engineering and Music*, 2001, pp. 632–644.

[4] Sergi Jordà, "Digital Lutherie: Crafting musical computers for new musics' performance and improvisation", PhD thesis, Universitat Pompeu Fabra, Departament de Tecnologia, 2005.

[5] Stephen R Wilson, "Music Sampling Lawsuits: Does Looping Music Samples Defeat the De Minimis Defense?", *Journal of High Technology Law*, vol. 1, pp. 179–193, 2002.

[6] The Inflation Calculator, "What cost $29000 in 1979...", Webpage, (uses the annual Statistical Abstracts of the United States), Nov. 14, 2014. URL: `http://www.westegg.com/inflation/infl.cgi?money=29000&first=1979&final=2013` (visited on 11/14/2014).

[7] Gordon Earle Moore, "Cramming more components onto integrated circuits", *Electronics Magazine*, Apr. 19, 1965.

[8] Andrew Huang, *Hacking the Xbox: An introduction to reverse engineering*. No Starch Press, 2003.

[9] Curtis Roads and Max Mathews, "Interview with Max Mathews", *Computer Music Journal*, vol. 4, no. 4, pp. 15–22, 1980.

[10] Kevin W Leary, "The personal sound system", in *Electro/94 International. Conference Proceedings. Combined Volumes.*, IEEE, 1994, pp. 299–303.

[11] Nils Dittbrenner, *Soundchip-Musik: Computer-und Videospielmusik von 1977-1994*. Osnabrück, Germany: epOs Music, 2007, ISBN: 978-3-923486-94-6.

[12] Morten Sieker Andreasen, Henrik Villemann Nielsen, Simon Ormholt Schrøder, and Jan Stage, "Usability in open source software development: opinions and practice", *Information technology and control*, vol. 35, no. 3A, pp. 303–312, 2006, ISSN: 1392-124X.

[13] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey, "How do programmers ask and answer questions on the web? (NIER track)", in *Software Engineering (ICSE), 2011 33rd International Conference on*, IEEE, 2011, pp. 804–807.

[14] Kawin Ngamkajornwiwat, Dongsong Zhang, Akif Güneş Koru, Lina Zhou, and Robert Nolker, "An exploratory study on the evolution of OSS developer communities", in *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, IEEE, 2008, pp. 305–305.

[15] Andrew Begel, Jan Bosch, and Margaret-Anne Storey, "Social networking meets software development: Perspectives from GitHub, MSDN, Stack Exchange, and TopCoder", *Software, IEEE*, vol. 30, no. 1, pp. 52–66, 2013.

[16] Steven Harris, Steven Green, and Ka Leung, "Techniques to Measure and Maximize the Performance of a 120 dB, 24-Bit, 96-kHz A/D Converter Integrated Circuit", in *Audio Engineering Society Convention 103*, Audio Engineering Society, Sep. 1997. URL: `http://www.cirrus.com/en/pubs/whitePaper/aes04.pdf`.

[17] Dr. Steven Harris and Clif Sanchez, "Personal Computer Audio Quality Measurements", Cirrus Logic, white paper, Mar. 1999. URL: http://www.cirrus.com/en/pubs/whitePaper/meas100.pdf.

[18] Stevan Harnad, Tim Brody, François Vallières, Les Carr, Steve Hitchcock, Yves Gingras, Charles Oppenheim, Heinrich Stamerjohanns, and Eberhard R Hilf, "The access/impact problem and the green and gold roads to open access", *Serials review*, vol. 30, no. 4, pp. 310–314, 2004.

[19] Scott F Aikin, "Poe's Law, Group Polarization, and the Epistemology of Online Religious Discourse", ser. Working papers, Social Science Research Network, Jan. 23, 2009. DOI: 10.2139/ssrn.1332169.

[20] Carlo Nardi, "Performing electronic dance music: mimesis, reflexivity and the commodification of listening", *Contemporanea - Revista de Comunicação e Cultura*, vol. 10, no. 1, pp. 80–98, 2012.

[21] Michael Kanellos, "Moore's Law to Conk in 10, 15 Years, Says Moore", *CNET News*, vol. 18, Sep. 18, 2007. URL: http://www.cnet.com/news/moores-law-to-conk-in-10-15-years-says-moore/ (visited on 11/21/2014).

[22] Ed Montano, "'How do you know he's not playing Pac-Man while he's supposed to be DJing?': technology, formats and the digital future of DJ culture", *Popular Music*, vol. 29, no. 03, pp. 397–416, 2010.

[23] Jim Highsmith and Alistair Cockburn, "Agile software development: The business of innovation", *Computer*, vol. 34, no. 9, pp. 120–127, 2001.

[24] Gary Sperrazza, "Looka Here! It's Sam & Dave!", *Time Barrier Express*, vol. 3, no. 26, pp. 18–31, Sep. 1979.

[25] Arthur Charles Clarke, "Hazards of prophecy: the failure of imagination", in *Profiles of the future: An inquiry into the limits of the possible. Rev. ed*, Harper & Row, 1973.

[26] Julijana Žabeva Papazova, "Alternative Rock Music in Yugoslavia in the Period Between 1980-1991 and its Influence on the Present Musical and Cultural Life in Macedonia, Serbia and Croatia", *IASPM@ Journal*, vol. 4, no. 1, pp. 117–119, 2014.

[27] Biljana Stojanović, "Exchange Rate Regimes of the Dinar 1945–1990: An Assessment of Appropriateness and Efficiency", in *Proceedings of OeNB Workshops: The Experience of Exchange Rate Regimes in Southeastern Europe in a Historical and Comparative Perspective*, Oesterreichische Nationalbank, Apr. 13, 2007.

[28] CIA, *The world factbook 1995*. Washington D.C.: Central Intelligence Agency, 1995.

[29] filtrov, "Dead Cops Rock@CODEX (Skopje, 1992) - YouTube", video, Oct. 24, 2008. URL: https://www.youtube.com/watch?v=a6obLRZLqGY (visited on 12/02/2014).

[30] John WC Van Bogart, "Magnetic tape storage and handling", *A Guide for Libraries and Archives, National Media Laboratory*, 1995.

[31] Ljupčo Jolevski, "Ritmistica", *Блесок - литература и други уметности (Shine - literature & other arts)*, no. 17, 2000.

[32] Kjetil Falkenberg Hansen, "The acoustics and performance of DJ scratching, Analysis and modelling", PhD thesis, KTH, Music Acoustics, 2010, pp. xii, 74, ISBN: 978-91-7415-541-9.

[33] Bill Werde, "The DJ's new mix: Digital files and a turntable", *The New York Times*, 2001.

[34] Smilen Dimitrov, ""DMX Director" - Architecture of a 3D light-programming application, in a multi-user Internet environment", Aalborg University Copenhagen, Copenhagen, Denmark, Final report (M.Sc. thesis equivalent), May 2006, p. 113.

[35] Ableton, "Ableton homepage", webpage. URL: `https://www.ableton.com/` (visited on 02/05/2015).

[36] Miller Puckette, "Pure Data: another integrated computer music environment", in *Proceedings of the Second Intercollege Computer Music Concerts*, Tachikawa, Japan, 1996, pp. 37–41.

[37] Joseph A Sarlo, "GrIPD: A Graphical Interface Editing Tool and Run-time Environment for Pure Data", in *Proceedings of the International Computer Music Conference*, International Computer Music Association, 2003, p. 305.

[38] Smilen Dimitrov, "PhD thesis website", 2007. URL: `http://imi.aau.dk/~sd/phd` (visited on 2015).

[39] Vangel Nonevski, *Грамофонот како метаинструмент: од механичка репродукција до ремикс култура. The Turntable as a metainstrument: From mechanical reproduction to remix culture*, Macedonian and English. Skopje, Macedonia: Аксиома (Aksioma), 2014, p. 215, ISBN: 978-608-65729-0-7.

[40] Dan Stowell and Alex McLean, "Live music-making: A rich open task requires a rich open interface", in *Music and human-computer interaction*, Springer, 2013, pp. 139–152.

[41] Jamie Zigelbaum, Amon Millner, Bella Desai, and Hiroshi Ishii, "BodyBeats: whole-body, musical interfaces for children", in *CHI'06 Extended Abstracts on Human Factors in Computing Systems*, ACM, 2006, pp. 1595–1600.

[42] Matthew John Yee-King, "The evolving drum machine", in *Music-AL workshop, ECAL conference*, vol. 2007, 2007.

[43] Yee Chieh Denise Chew and Eric Caspary, "MusEEGk: a brain computer musical interface", in *CHI'11 Extended Abstracts on Human Factors in Computing Systems*, ACM, 2011, pp. 1417–1422.

[44] Martin Russ, *Sound Synthesis and Sampling*. Taylor & Francis, 2012, ISBN: 978-1-136-12214-9.

[45] Mark Vail, *The Synthesizer: A Comprehensive Guide to Understanding, Programming, Playing, and Recording the Ultimate Electronic Music Instrument*. Oxford University Press, USA, 2014, ISBN: 978-0-19-539489-4.

[46] Sofia Dahl, "On the beat : human movement and timing in the production and perception of music", QC 20101004, PhD thesis, KTH, Speech, Music and Hearing, TMH, 2005, pp. x, 77.

[47] Marco Fabiani, "Interactive computer-aided expressive music performance: Analysis, control, modification and synthesis", PhD thesis, KTH, Music Acoustics, 2011, p. 69, ISBN: 978-91-7501-031-1.

[48] Johannes Kreidler, *Loadbang : Programming electronic music in Pd*. Hofheim: Wolke, 2009, ISBN: 978-3-936000-57-3.

[49] Miller Puckette, *The Theory and Technique of Electronic Music*. Singapore: World Scientific Publishing Company, 2007, ISBN: 978-981-270-077-3.

[50] David Arditi, "Digital Downsizing: The Effects of Digital Music Production on Labor", *Journal of Popular Music Studies*, vol. 26, no. 4, pp. 503–520, 2014, ISSN: 1533-1598. DOI: 10.1111/jpms.12095.

[51] William W Gaver, "What in the world do we hear?: An ecological approach to auditory event perception", *Ecological psychology*, vol. 5, no. 1, pp. 1–29, 1993.

[52] Simon Crab, "120 Years of Electronic Music", 1996. URL: http://120years.net/ (visited on 02/10/2015).

[53] Various authors, "Wikimedia Commons", used files: Roland_TR-808_drum_machine.jpg, 111607sp1200.jpg, YAMAHA_RY30.JPG, Akai_MPC2000XL_front.jpg, Technics_SL-1200MK2-2.jpg, Turntables_and_mixer.jpg, Douris_Man_with_wax_tablet.jpg. URL: http://commons.wikimedia.org/wiki/ (visited on 02/09/2015).

[54] Gordon Reid, "Practical Snare Drum Synthesis", *Sound On Sound Magazine*, Synth Secrets, Apr. 2002.

[55] "Synth School, Part 5: The Origins Of S&S", *Sound On Sound Magazine*, Synth School, Dec. 1997.

[56] dmcdjchamps.com, "Technics SL Series... R.I.P.", Nov. 1, 2010. URL: http://www.dmcdjchamps.com/news-view.php?n=Mjk5 (visited on 02/16/2015).

[57] Chris Supranowitz, "General Topics in Electron Microscopy: Micrograph Acquisition, Post-Processing, Special Techniques (OPT307)", Jan. 24, 2013. URL: http://www.optics.rochester.edu/workgroups/cml/opt307/spr05/chris/ (visited on 02/16/2015).

[58] Sylvain Stotzer, "Phonographic record sound extraction by image processing", PhD thesis, Faculty of Science, University of Fribourg (Switzerland, 2006.

[59] Beiming Wang and Mark D Plumbley, "Musical audio stream separation by non-negative matrix factorization", in *Proceedings of the Digital Music Research Network (DMRN) Summer Conference*, 2005, pp. 23–24.

[60] Romain Hennequin, Bertrand David, and Roland Badeau, "Score informed audio source separation using a parametric model of non-negative spectrogram", in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2011.

[61] Pierre Schaeffer, "Acousmatics", *Audio culture: Readings in modern music*, pp. 76–81, 2004.

[62] Kurt B Reighley, *Looking for the Perfect Beat: The Art and Culture of the DJ*. MTV Books, 2000, ISBN: 978-0-671-03869-4.

[63] Jean Laroche and Mark Dolson, "New phase-vocoder techniques for real-time pitch shifting, chorusing, harmonizing, and other exotic audio modifications", *Journal of the Audio Engineering Society*, vol. 47, no. 11, pp. 928–936, 1999.

[64] Jordi Bonada, "Automatic technique in frequency domain for near-lossless time-scale modification of audio", in *Proceedings of International Computer Music Conference*, 2000, pp. 396–399.

[65] Mark Dolson, "The phase vocoder: A tutorial", *Computer Music Journal*, pp. 14–27, 1986.

[66] Kjetil Falkenberg Hansen, Marco Fabiani, and Roberto Bresin, "Analysis of the acoustics and playing strategies of turntable scratching", *Acta Acustica united with Acustica*, vol. 97, no. 2, pp. 303–314, 2011.

[67] Jean-Claude Chuzeville (Director), "Bruit est Musique", Video documentary, JPL Productions and TL7, France, 2010.

[68] Peter Desain and Henkjan Honing, "Tempo curves considered harmful", *Contemporary Music Review*, vol. 7, no. 2, pp. 123–138, 1993.

[69] Peter Kirn, "NI Ends Legal Dispute Over Traktor Scratch; Digital Vinyl's Twisty, Turny History", Apr. 28, 2008. URL: `http://createdigitalmusic.com/2008/04/ni-ends-legal-dispute-over-traktor-scratch-digital-vinyls-twisty-turny-history/` (visited on 02/18/2015).

[70] Takuro Mizuta Lippit, "Turntable Music in the Digital Era: Designing Alternative Tools for New Turntable Expression", in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Norbert Schnell, Frédéric Bevilacqua, Michael Lyons, and Atau Tanaka, Eds., Paris, France, 2006, pp. 71–74.

[71] William M Hartmann, "The electronic music synthesizer and the physics of music", *Am. J. Phys*, vol. 43, p. 755, 1975.

[72] James T Kajiya, "The rendering equation", in *ACM Siggraph Computer Graphics*, ACM, vol. 20, 1986, pp. 143–150.

[73] Valerii Salov, "Notation for Iteration of Functions, Iteral", *ArXiv preprint, arXiv: 1207.0152*, 2012.

[74] Mike J Potel, "Real-time playback in animation systems", in *ACM SIGGRAPH Computer Graphics*, ACM, vol. 11, 1977, pp. 72–77.

[75] Thomas Grill, "py/pyext – Python scripting objects for Pure Data and Max". URL: `http://grrrr.org/research/software/py/` (visited on 02/22/2015).

[76] Tue H. Andersen, "Mixxx : Towards Novel DJ Interfaces", in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Marcelo M. Wanderley, Richard McKenzie, and Louise Ostiguy, Eds., Montreal, 2003, pp. 30–35.

[77] Takuro M. Lippit, "Realtime Sampling System for the Turntablist, Version 2: 16padjoystickcontroller", in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Yoichi Nagashima, Yasuo Ito, and Yuji Furuta, Eds., Hamamatsu, Japan, 2004, pp. 211–212.

[78] Nikita Pashenkov, "A New Mix of Forgotten Technology: Sound Generation, Sequencing and Performance Using an Optical Turntable", in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Yoichi Nagashima, Yasuo Ito, and Yuji Furuta, Eds., Hamamatsu, Japan, 2004, pp. 64–67.

[79] Kentaro Fukuchi, "Multi-track scratch player on a multi-touch sensing device", in *Entertainment Computing–ICEC 2007*, Springer, 2007, pp. 211–218.

[80] Pedro Lopez, Alfredo Ferreira, and J. A. Madeiras Pereira, "Battle of the DJs: an HCI Perspective of Traditional, Virtual, Hybrid and Multitouch DJing", in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Alexander R. Jensenius, Anders Tveit, Rolf I. Godoy, and Dan Overholt, Eds., Oslo, Norway, 2011, pp. 367–372.

[81] Karl Yerkes, Greg Shear, and Matthew Wright, "Disky : a DIY Rotational Interface with Inherent Dynamics", in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Kirsty Beilharz, Bert Bongers, Andrew Johnston, and Sam Ferguson, Eds., Sydney, Australia, 2010, pp. 150–155.

[82] Steven Gelineck and Stefania Serafin, "A quantitative evaluation of the differences between knobs and sliders", in *New Interfaces for Musical Expression*, 2009.

[83] Steven Gelineck, "Exploratory and Creative Properties of Physical-Modeling-based Musical Instruments: Developing a framework for the development of physical modeling based digital musical instruments, which encourage exploration and creativity", PhD thesis, Department of Architecture, Design & Media Technology, Aalborg University, 2012.

[84] Niels Böttcher, "Procedural audio for computer games with motion controllers – Evaluating the design approach and investigating the player's perception of the sound and possible influences on the motor behaviour", PhD thesis, Department of Architecture, Design & Media Technology, Aalborg University, 2014.

[85] Timothy Beamish, Kees Van Den Doel, Karon MacLean, Sidney Fels, *et al.*, "D'groove: A haptic turntable for digital audio control", in *Proc. of the International Conference on Auditory Display*, Boston, MA, 2003.

[86] Matthew Wright, "Open sound control: an enabling technology for musical networking", *Organised Sound*, vol. 10, no. 03, pp. 193–200, 2005.

[87] Nicolas Villar, Adam T. Lindsay, and Hans Gellersen, "Pin & Play & Perform: A rearrangeable interface for musical composition and performance", in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Sidney S. Fels, Tina Blaine, Andy Schloss, and Sergi Jord'a, Eds., Vancouver, BC, Canada, 2005, pp. 188–191.

[88] Katharina Vogt, "Sonification of simulations in computational physics", PhD thesis, Institute for Electronic Music, Acoustics, University of Music, and Performing Arts, Graz, Austria, 2010.

[89] Kevin Xiaoguo Zhu and Zach Zhizhong Zhou, "Lock-In Strategy in Software Competition: Open-Source Software vs. Proprietary Software", *Information Systems Research*, vol. 23, no. 2, pp. 536–545, 2012.

[90] Amy Kucharik, "Vendor lock-in, part 1: Proprietary and lock-in not necessarily synonymous", Jul. 10, 2003. URL: `http://searchenterpriselinux.techtarget.com/news/913129/Vendor-lock-in-part-1Proprietary-and-lock-in-not-necessarily-synonymous` (visited on 02/25/2015).

[91] Richard J Gilbert, "Networks, Standards, and the Use of Market Dominance: Microsoft (1995)", *The Antitrust Revolution: The Role of Economics*, vol. 3, 1998.

[92] Anne-Kathrin Kuehnel, "Microsoft, Open Source and the software ecosystem: of predators and prey—the leopard can change its spots", *Information & Communications Technology Law*, vol. 17, no. 2, pp. 107–124, 2008. DOI: 10.1080/13600830802204229.

[93] Microsoft Corporation, "Shared Source Initiative", 2015. URL: http://www.microsoft.com/en-us/sharedsource/default.aspx (visited on 02/25/2015).

[94] Clifford Adelman, "A parallel universe: Certification in the information technology guild", *Change: The Magazine of Higher Learning*, vol. 32, no. 3, pp. 20–29, 2000.

[95] Joel West, "How open is open enough?: Melding proprietary and open source platform strategies", *Research policy*, vol. 32, no. 7, pp. 1259–1285, 2003.

[96] Chip Chapin, "CD-DA (Digital Audio) 1", Apr. 16, 2005. URL: http://www.chipchapin.com/CDMedia/cdda1.php3 (visited on 02/24/2015).

[97] Ken C. Pohlmann, *The Compact Disc: A Handbook of Theory and Use*, ser. Computer Music and Digital Audio Series. A-R Editions, 1989, ISBN: 978-0-89579-228-0.

[98] John Watkinson, *The Art of Digital Audio*, 3rd ed. Focal Press, Taylor & Francis Group, 2001, ISBN: 978-0-240-51587-8.

[99] Teruo Muraoka, Yoshihiko Yamada, and Masami Yamazaki, "Sampling-frequency considerations in digital audio", *Journal of the Audio Engineering Society*, vol. 26, no. 4, p. 252, 1978.

[100] Knut Blind, "Patent pools-a solution to patent conflicts in standardisation and an instrument of technology transfer: the MP3 case", in *Standardization and Innovation in Information Technology, 2003. The 3rd Conference on*, IEEE, Oct. 2003, pp. 27–35. DOI: 10.1109/SIIT.2003.1251192.

[101] Chris Bryden, "nanobox linux: nanobox-smb - Single Disk Windows Networking Client", Apr. 10, 2010. URL: http://web.archive.org/web/20100410143654/http://www.neonbox.org/nanobox/index.html (visited on 03/03/2015).

[102] Burton Grad, "A personal recollection: IBM's unbundling of software and services", *Annals of the History of Computing, IEEE*, vol. 24, no. 1, pp. 64–71, Jan. 2002, ISSN: 1058-6180. DOI: 10.1109/85.988583.

[103] Steven Levy, *Hackers: Heroes of the Computer Revolution*, 1st ed. New York, NY, USA: Doubleday, 1984, ISBN: 0-385-19195-2.

[104] Richard M. Stallman, *Free Software, Free Society: Selected Essays of Richard M. Stallman*, 2nd ed. Joshua Gay, Ed., revisor Free Software Foundation (Cambridge, Mass.), with a forew. by Lawrence Lessig. Boston, MA, USA: GNU Press, 2010, ISBN: 978-0-9831592-0-9.

[105] Vladimir Ilyich Lenin, *Империализм как высшая стадия капитализма. Imperialism : The last stage of capitalism*, Russian and English. London: Communist Party of Great Britain, 1917, p. 159.

[106] Donald Joy, "Unemployment Checks Forever: Politicians Pandering To Parasites", Jan. 8, 2014. URL: http://clashdaily.com/2014/01/unemployment-checks-forever-politicians-pandering-parasites/ (visited on 03/13/2015).

[107] Sgouris Sgouridis, "Defusing the energy trap: the potential of energy-denominated currencies to facilitate a sustainable energy transition", *Frontiers in Energy Research*, vol. 2, no. 8, p. 12, 2014, ISSN: 2296-598X. DOI: 10.3389/fenrg.2014.00008.

[108] Joseph Roisman, *Ancient Greece from Homer to Alexander: The Evidence*, trans. by John C. Yardley, ser. Blackwell Sourcebooks in Ancient History. John Wiley & Sons, 2011, ISBN: 978-1-4051-2775-2. URL: https://books.google.dk/books?id=krnW1m-kpvoC.

[109] Alfred Wassink, "Inflation and Financial Policy under the Roman Empire to the Price Edict of 301 A.D.", *Historia: Zeitschrift für Alte Geschichte*, vol. 40, no. 4, pp. 465–493, 1991, ISSN: 00182311.

[110] Andrew Ó Baoill, "The Significance of the World's First Copyright Ruling for Contemporary Debate on Intellectual Property", Paper presented at the annual meeting of the International Communication Association, Dresden International Congress Centre, Dresden, Germany, Jun. 16, 2006.

[111] K. Eric Drexler and Richard E. Smalley, "Drexler and Smalley make the case for and against 'molecular assemblers'", with an intro. by Rudy Baum, *Chemical & Engineering News*, vol. 81, no. 48, p. 1, 2003.

[112] José López, "Bridging the gaps: science fiction in nanotechnology", *HYLE — International Journal for Philosophy of Chemistry*, vol. 10, no. 2, pp. 129–152, 2004.

[113] Courtney Love, "Courtney Love does the math", *Salon.com*, vol. 14, Jun. 14, 2000. URL: http://www.salon.com/2000/06/14/love_7/ (visited on 03/19/2015).

[114] Philip Cregan, "What Are the Effects Of Illegal Downloading On The Music Industry?", B.A. Thesis, Dublin, National College of Ireland, 2011.

[115] Lincoln D Durey, "EOF: dear laptop vendor", *Linux Journal*, vol. 2004, no. 126, p. 13, 2004.

[116] Thomas Andersen, "The Norwegian DeCSS Litigation–A DVD Piracy Trial", *Business Law Review*, vol. 25, no. 7, pp. 187–189, 2004.

[117] Chris Hoffman, "Why Watching DVDs on Linux is Illegal in the USA", Mar. 1, 2013. URL: http://www.howtogeek.com/138969/why-watching-dvds-on-linux-is-illegal-in-the-usa/ (visited on 03/19/2015).

[118] CNNMoney (New York), "How the NSA can 'turn on' your cell phone remotely", Jun. 6, 2014. URL: http://money.cnn.com/2014/06/06/technology/security/nsa-turn-on-phone/ (visited on 03/19/2015).

[119] Joel Hruska, "Windows 8 phones home, tells Microsoft every time you install a program", Aug. 24, 2012. URL: http://www.extremetech.com/computing/135010-windows-8-phones-home-tells-microsoft-every-time-you-install-a-program (visited on 03/19/2015).

[120] Cory Doctorow, "GM says you don't own your car, you just license it", May 21, 2015. URL: http://boingboing.net/2015/05/21/gm-says-you-dont-own-your-ca.html (visited on 05/27/2015).

[121] Joan Archer, Morgan Chu, Karen Robinson, and Hon James S Ware, "Apple v. Samsung Design Patents Take Center Stage?", in *ABA Section of Litigation, ABA Annual Meeting*, American Bar Association, San Francisco, CA, Aug. 2013.

[122] James E Bessen, Michael J Meurer, and Jennifer Laurissa Ford, "The private and social costs of patent trolls", *Boston Univ. School of Law, Law and Economics Research Paper*, no. 11-45, 2011.

[123] Maria Kechagia, Diomidis Spinellis, and Stephanos Androutsellis-Theotokis, "Open Source Licensing Across Package Dependencies", in *Informatics (PCI), 2010 14th Panhellenic Conference on*, Sep. 2010, pp. 27–32. DOI: 10.1109/PCI.2010.28.

[124] Eric S. Raymond, *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media, 2001, ISBN: 978-0-596-55396-8.

[125] Nick Valery, "The Difference Engine: Linux's Achilles heel", Nov. 12, 2010. URL: http://www.economist.com/blogs/babbage/2010/11/operating_systems (visited on 03/21/2015).

[126] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea, "Testing Closed-Source Binary Device Drivers with DDT", in *USENIX Annual Technical Conference*, 2010.

[127] Vitaly Chipounov and George Candea, "Reverse-Engineering Drivers for Safety and Portability.", in *HotDep 08 - Fourth Workshop on Hot Topics in System Dependability*, San Diego, CA, Dec. 7, 2008.

[128] Smilen Dimitrov, "attenload - fetch data from Atten oscilloscopes - README", Feb. 16, 2013. URL: http://sdaaubckp.sourceforge.net/attenload/ (visited on 03/21/2015).

[129] Chun-Hui Miao, "Tying, Compatibility and Planned Obsolescence", *The Journal of Industrial Economics*, vol. 58, no. 3, pp. 579–606, 2010.

[130] Dejan Viduka and Ana Bašić, "Impact of Open Source software on the environmental protection", *Computational Ecology & Software*, vol. 5, no. 1, 2015.

[131] Ron Amadeo, "Google's iron grip on Android: Controlling open source by any means necessary", *Ars Tecnica*, vol. 21, 2013.

[132] Mark Doran, "The growing role of UEFI secure boot in Linux distributions", *Linux Journal*, vol. 2014, no. 239, p. 3, 2014.

[133] Meredith L. Patterson, "How I Explained Heartbleed To My Therapist — Riding Open Source's Race to the Bottom", Sep. 25, 2014. URL: https://medium.com/message/how-i-explained-heartbleed-to-my-therapist-4c1dbcbe1099 (visited on 03/17/2015).

[134] Susan A. McDaniel and Zachary Zimmer, *Global Ageing in the Twenty-First Century: Challenges, Opportunities and Implications*. Ashgate Publishing Limited, 2013, ISBN: 978-1-4724-0005-5.

[135] Satoshi Nakamoto (pseudonym), "Bitcoin: A peer-to-peer electronic cash system", p. 9, 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on 03/21/2015).

[136] R. T. Leuchtkafer (pseudonym), "High Frequency Trading: A bibliography of evidence-based research", Mar. 2015.

[137] Thomas Bourke, "Bibliography of the Global Financial / Economic Crisis", Jan. 2015.

[138] Greg Hill and Kerry Wendell Thornley, *Principia Discordia*. Loompanics Unlimited, 1979.

[139] Stefan Baumgärtner, "Thermodynamics of waste generation", *Waste in Ecological Economics, Edward Elgar, Cheltenham, Northampton*, 2002.

[140] Arthur Bloch, *Murphy's Law*. Perigee, 2003, ISBN: 978-0-399-52930-6.

[141] Ben Einstein, "Hardware by the Numbers (Part 1: Team + Prototyping)", Oct. 31, 2014. URL: https://medium.com/@BoltVC/hardware-by-the-numbers-part-1-team-prototyping-b225a33f55bf (visited on 03/25/2015).

[142] Kevin Menard, "Open Sourcing a Failed Startup", Nov. 20, 2014. URL: http://nirvdrum.com/2014/11/20/open-sourcing-mogotest.html (visited on 03/25/2015).

[143] Bharat Rao, Bojan Angelov, and Oded Nov, "Fusion of Disruptive Technologies: Lessons from the Skype Case", *European Management Journal*, vol. 24, no. 2, pp. 174–188, 2006.

[144] Nyle Steiner (K7NS), "Iron Pyrites Negative Resistance (RF) Oscillator", Feb. 22, 2001. URL: http://www.sparkbangbuzz.com/els/iposc-el.htm (visited on 02/24/2015).

[145] Hermann Ludwig Ferdinand von Helmholtz, *Die Lehre von den Tonempfindungen als physiologische Grundlage für die Theorie der Musik. On the Sensations of Tone as a Physiological Basis for the Theory of Music*, German and English, trans. by Alexander John Ellis. London: Longmans, Green and Co., 1875.

[146] Gareth Loy, "Musicians make a standard: the MIDI phenomenon", *Computer Music Journal*, vol. 9, no. 4, pp. 8–26, 1985.

[147] Erich Neuwirth, "Musical abstractions and programming paradigms", in *Proceedings of the 11th European Logo Conference*, Bratislava, Slovakia, 2007, ISBN: 978-80-89186-20-4.

[148] Roger B. Dannenberg, "The Interpretation of MIDI Velocity", in *Proceedings of the International Computer Music Conference (ICMC)*, San Francisco, CA: The International Computer Music Association, 2006, pp. 193–196.

[149] ——, "Music Representation Issues, Techniques, and Systems", *Computer Music Journal*, vol. 17, no. 3, pp. 20–30, 1993.

[150] Curtis Roads, *Microsound*. MIT Press, 2004, ISBN: 978-0-262-68154-4.

[151] Barbara Bryner, "The Piano Roll: A Valuable Recording Medium of the Twentieth Century", Master's thesis, Department of Music, University of Utah, 2002.

[152] Lisa Gitelman, "Media, Materiality, and the Measure of the Digital; or, the Case of Sheet Music and the Problem of Piano Rolls", *Memory bytes: History, technology, and digital culture*, pp. 199–217, 2004.

[153] Peter Manning, *Electronic and Computer Music*. Oxford University Press, USA, 2013, ISBN: 978-0-19-991259-9.

[154] Stefania Serafin, "The sound of friction: real-time models, playability and musical applications", PhD thesis, Stanford University, CCRMA, 2004.

[155] Karlheinz Stockhausen and Elaine Barkin, "The concept of unity in electronic music", *Perspectives of New Music*, vol. 1, no. 1, pp. 39–48, 1962.

Appendix A

Basic theoretical aspects of the classic rhythm/drum machine step sequencer

Here, the classic rhythm/drum machines (as exemplified by Roland TR-808 (1980), E-mu SP-1200 (1987), Yamaha RY30 (1991), and Akai MPC2000 (1997), shown on Fig. 3.1 in section 3.1) are in focus. In particular, the formulation of the unique live interaction characteristics of these machines will be undertaken, by establishing some elementary relationships between the operation of these machines - and concepts in basic music theory and music technology.

The facilities found in these machines for programming rhythms, can be said to be based in Western musical composition tradition – and in the concept of a musical note, the tonal aspect of which has been formally described mathematically since at least Helmholtz (1875, [145]). The musical note, as a sign, records the pitch and the duration of a sound in a single symbol; a simple sequence of notes can thus transcribe a monophonic melody, as played for instance on a flute. The MIDI specification [146] implements this abstraction as a part of an electronic control protocol; however, the note's duration, and thus the note, is not encoded as a single event [147], but as two - through the commands/messages `note on` and `note off`. On the other hand, for both note messages, MIDI specifies a *velocity* argument, whose meaning refers to the speed of a piano keyboard key press (reaching from depressed to pressed position) - but whose actual implementation is not specified [148], and is most often mapped to a note's loudness/volume; something which in traditional score notation is not a property of a note, but is instead indicated by words or alphabetic symbols such as ***p*** (*piano*) or ***f*** (*forte*), as dynamics directives. In spite of this and other confusions regarding MIDI as a music representation tool [149], some of the terminology that evolved around related hardware and software can be used to discuss the affordances of the drum machines.

Note that for pure (sinusoidal) tones, the frequency of the sinusoid gives rise to the sensation of pitch; Roads (2004, [150]) notes a perceptual threshold for metered rhythm as frequencies 8 Hz and below, for classical pitch sensation as frequencies above 40 Hz, and a "zone of ambiguity" of a continuous tone perception in-between. For complex sounds, pitch may be determined by the fundamental and overtone frequencies of the sound – or in general, its spectrum. In essence, both classical notation and MIDI tools preserve: the perspective on music composition as a two-dimensional layout, where the horizontal abscissa represents time and the vertical ordinate represents pitch/frequency; and the approach to generally quantizing the pitch at semitones of the twelve-tone, equal temperament, scale – however, as shown on Fig. AA.1, the visual correspondence between the two may not be exact.

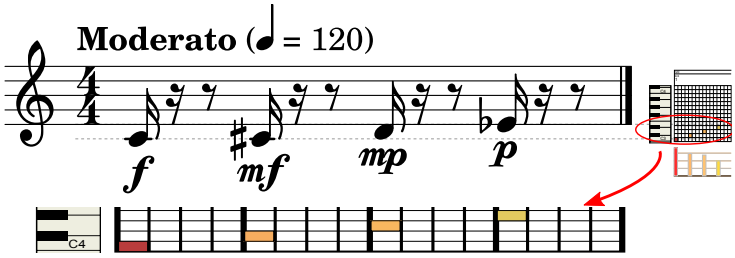


Fig. AA.1: Visual comparison between classical notation score, and software (here Rosegarden) interpretation and rendering as "piano roll" MIDI notes (velocities shown separately on the rightmost part, otherwise indicated by shading)

This is the case, at least because of the use of accidental notes in classical notation: modifiers like ♯ (sharp, *dièse*) or ♭ (flat, *bemolle*) change the pitch of the note written thereafter, causing different pitches to be notated at the same vertical coordinate on the staff; conversely, the MIDI event notation is explicit in allocating an exclusive vertical coordinate to each semitone pitch.

In terms of playback speed or *tempo*, for a great part of history classical notation used descriptive words (e.g. *Moderato*) to denote the speed at which a composition should be played. Only after the increased popularity of metronomes, a convention arose where a type of a note (e.g. a quarter note ♩ a.k.a. crotchet) is declared to represent one *beat*, and its duration is expressed in terms of BPM - inscribed, as a metronomic indication, in parentheses (♩ = 120) next to the traditional tempo description. MIDI inherits this tradition, in that it defines a MIDI meta event message - one that may exist in files, but is not transmitted to devices - called `set tempo`, which encodes the duration of a quarter note in microseconds (but which is displayed as BPM in MIDI editors). If d_q denotes the duration of a quarter note (a beat) in seconds, and M_T denotes the metronome tempo in BPM, then the relation between two can be formally expressed with

Eq. AA.1.

$$d_q = \frac{1}{M_T[\frac{beats}{min}]} = \frac{1\,[min]}{M_T\,[beats]} = \frac{60}{M_T}\,[\frac{sec}{beat}] \tag{AA.1}$$

Thus per Eq. AA.1, a metronome tempo of 120 BPM indicates a beat duration of 0.5 seconds, or 500 µs; if [beat] is accepted as a dimensionless unit, then BPM as a physical unit – as [1/time] – would be equivalent to the unit of frequency, Hertz [Hz] (albeit with different numeric values). Generally, Eq. AA.1 specifies the duration of a beat (d_b), which most commonly is the duration of a quarter note (d_q), but depending on the tempo definition, could be set to other note types, like half or 16th note.

When we talk about loops in context of classic sequencers, it should be noted that by default, we refer to loops of duration of a single *measure* (or bar) as a discrete unit of time. In classical music notation, the duration of a measure is expressed through the time signature: for instance, the time signature $\frac{4}{4}$ refers to the duration of four (4) quarter (1/4) notes as the duration of a measure in that composition; Fig. AA.1 depicts one such measure. Therefore, as per Eq. AA.1, a single $\frac{4}{4}$ measure at 120 BPM would last for two seconds – and this is the general order of magnitude of time periods considered, when talking about repetition of audio loops in this scope.

It might be useful at this point, to introduce a term that may aid the formalization efforts here: let a *note lane* represent a collection (a sequence) of notes, all at the same pitch. While useless in terms of melodic, even monophonic, music - it can be used to refer to a rhythm played by a single percussive instrument, an example of which is shown on Fig. AA.2.

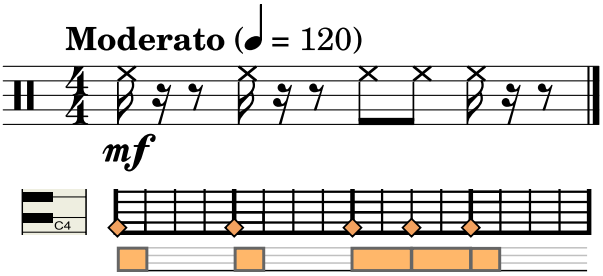


Fig. AA.2: Visual comparison between classical percussion score (using a hi-hat symbol), and software (here Rosegarden) interpretation and rendering as "piano roll" MIDI percussion (note the duration is only shown through the velocities, shown below)

The MIDI visualization shown on Figs. AA.1, AA.2 bottom is known as a "piano roll", stemming from the recording medium of the early reproducing or player piano technology [151, 152]. In both classical notation and piano-roll visualization, in context of percussion, the vertical coordinate stops being a notation for pitch, and instead is treated as an index into a set (or table) of percussive instrument sounds. This is appropriate for percussion instruments, as often they are perceived as non-pitched, and are incapable of producing

chords (that is, can only reproduce one sound event at a time); correspondingly, in electronic synthesis, sounds for snare drums and hi-hat have long been modelled by inclusion of noise generators [54] in the audio generating circuits. Classical notation, as on Fig. AA.2, may emphasize this with a special symbol standing in for the note head; while MIDI arrangement software may offer a "percussion" view into a sequence. In the MIDI percussion view, instead of with a rectangle with width indicating the note duration, the event may be visualized with a different symbol (a rhombic or "diamond" shape) only at the start (i.e. the *trigger*) point; while the duration might only be shown otherwise, for instance through the width of the velocities (e.g. as in the Rosegarden software). So, in cases of notation where we can consider the pitch information to be a general index into a set of instruments (wherein a single pitch corresponds to a single instrument sound) – a note lane would represent a sequence played by a single instrument (visually, it would be what is left of the piano roll on Fig. AA.2 bottom, if all space apart from the one corresponding to one key, say C4, is covered). A note lane could control a very simple electronic instrument engine, a model of which is shown on Fig. AA.3.

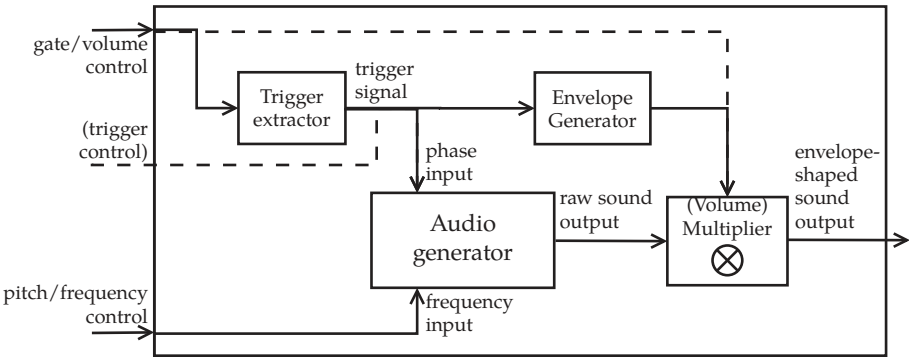


Fig. AA.3: A simple model of a basic electronic instrument sound generator.

The model on Fig. AA.3 is based on models, often considered basic in electronic music (cf. Fig. 1 in [71], Fig. 1.5 in [49], Fig. 32 in [153]). The control signals on Fig. AA.3 could be considered either as analog electronic signals (e.g. an electric voltage changing in time), or as digital signals (e.g. a stream of numbers, whose sequential order implies timing information). The audio generator could be either an analog electronic oscillator, a digital sound sampler, or embody any other synthesis method, e.g. based on physical modelling (refer to Serafin (2004, [154])). Note that analog oscillators, in principle, start oscillating as soon as, and for as long as, the circuit is powered; so instead of turning the oscillator on and off each time a note is played, the signal is typically ran through a device that functions as a multiplier (a.k.a. VCA), whose control input signal determines the volume of the final sound output. In principle, the "gate/volume control" signal can be conceptually considered as a rectangular function of

time in the range from 0 to 1, whose duration specifies the duration of a note event: multiplication with 0 then results with no sound on the output; and multiplication with 1 with the original, raw generator sound on the output. As such, a "gate/volume control" signal could be directly applied to the multiplier to control the final sound output (although, note that in MIDI, additional though straightforward processing is required, to obtain such a signal from `note on` and `note off` messages). However, considering that analog oscillators produce sound at a fixed volume, in analog electronics instruments the oscillator sound is typically shaped with an ADSR envelope – a piecewise linear signal generator. The envelope needs a starting signal, called the "trigger signal" on the Fig. AA.3, which in principle is synonymous with the start of the "gate/volume control" signal. As such, in can be derived from "gate/volume control" signal: if the "gate/volume control" rectangular signal is mathematically considered as a composition of two Heaviside step functions, then its differentiation in time would result with two Dirac function pulses $\delta(t)$; the positive pulse at the start of the "gate/volume control" would represent the "trigger control" signal. In analog electronics, this trigger extraction can be achieved with generic rising edge detector circuits, such as a linear phase comparator, or a (positive) edge-triggered one-shot (monostable multivibrator); in digital signal processing, it can be found simply by calculating the differential signal $x[t] - x[t-1]$ of successive values in time. However, the "trigger control" signal does not have to be derived from the "gate/volume control" signal - it can also be supplied independently.

If the audio generator on the model on Fig. AA.3 represents an analog oscillator, then the pitch control signal sets the frequency of oscillation of the circuit; however, if the audio generator is a digital sampler, then the pitch control would set the playback speed of the sound sample (through which the sensation of different pitches can be achieved with sound samples). The issue of phase is different: in an analog oscillator, the oscillation frequency f in Hertz, is related to the period T in seconds as per:

$$f = \frac{1}{T} \tag{AA.2}$$

Thus, a sinusoidal oscillator reproducing middle A (A4) at 440 Hz, has a period of 2.27 ms; since the start of the note is likely to be "covered" by the attack of the ADSR envelope lasting at least tens of milliseconds, in a monophonic context the phase (if the oscillation started relatively earlier or later in the range of the period $\pm$2.27 ms) won't matter much to a listener. On the other hand, if the audio generator is a sampler, reproducing sound samples with durations in range of seconds (e.g. timpani, cymbals, even whole rhythmic loops), it matters very much if the start of the note corresponds to the start, or say the middle, of the sound sample (which how phase expresses itself in this context). Thus, if the audio generating engine should reproduce sound samples of percussion, as a starting point we'd want the sound sample phase to be reset at each new start of a note – which is why the trigger signal is shown routed to the phase input

of the audio generator on Fig. AA.3. Finally, it should be noted that the model on Fig. AA.3 is applicable only to strictly monophonic operations: subsequent notes start by terminating the previous ones if they would otherwise still last, both in terms of pitch and in terms of duration (both with ADSR envelope and without); much like a flute or a single string on a violin. Thus, if we'd want two or more notes reproduced at the same time, as in a piano chord (i.e. polyphony) - we would, in principle, have to employ two or more generators as on Fig. AA.3, tuned to reproduce the same timbre.

Returning to the classic sequencers on Fig. 3.1, it is notable that all of them have interfaces quite rich with elements such as buttons; however, only some of those are of interest here. Let's first mention that step sequencing is defined as opposed to live, real-time sequencing: instead of recording the sequence by "drumming" it real-time on particular interface keys or pads, the user is expected to toggle (or choose) whether a note on a particular step in the measure sequence is playing or not. The most basic measure quantization setting on such drum machines is in 16^{th} notes (𝅘𝅥𝅯 or semiquaver); that would imply a measure in $\frac{16}{16}$ time signature. Note however that the duration of a $\frac{16}{16}$ measure is mathematically equal to the duration of a $\frac{4}{4}$ one; and in classical notation, the choice of time signature depends on the interpretation of accents in the work generally, not the smallest note duration as per the quantization of a measure (e.g. if the accent falls on each fourth 16^{th} note, that measure would be signed as $\frac{4}{4}$). Thus, it can be stated that:

- A *note lane* is a sequence of single-pitch notes at the quantization resolution of a measure, able to drive a single monophonic generator
- A *pattern* or *segment* is a stack of note lanes, attributed to either different pitches of a single instrument or different instrument sounds, with the length of at least a measure (can be equivocated to "piano roll" display); able to represent polyphony
- A *track* is a sequence of patterns, associated to a single instrument definition (either as a pitched instrument, or a collection of sounds such as a "drum bank" - which would correspond to the notion of a MIDI channel)
- A *song* is a stack of tracks with pre-programmed pattern sequences and predefined sound sets (or banks); able to represent multi-timbrality

In fact, this kind of hierarchical perception of sonic events has been noticed in electronic music theory at the least since Stockhausen *et al.* (1962, [155]), who noted that "... a musical composition is no more than a temporal ordering of sound events, just as each sound event in a composition is a temporal ordering of pulses. It is only a question of the point at which composition begins ..." [155]. Also, this terminology is close to the one used in the classic drum machines, as well as MIDI editors. Thus, the distinct features of the classic drum machines can be described as follows:

- **TR-808**: Fig. 3.1 a) indicates a row of 16 step buttons, representing a note lane of a measure quantized in 16^{th} notes; the user can press these to toggle the playback of a sound on a particular step while the sequence is playing, and thus change the sequence "live" (even if that sound will first be heard next time the sequence loops at that position). It is imaginable that a press slide across multiple buttons would result with a sort of a drum roll, and that it would be possible to toggle that roll from one measure to the other, allowing for real-time programming of some rhythmic transitions. Changing which sound the step buttons' row applies to, is achievable through other elements on the interface.

- **SP-1200**: Fig. 3.1 b) indicates a section that includes a row of 8 buttons; these however do not represent a note lane, or an interface for a step sequencer – rather, they can be approximated to a piano keyboard, except they trigger individual (typically drum) sound samples, and are intended for real-time programming of a drum sequence. A step sequencer is accessible by other means (left/right arrow buttons). Additionally, volume faders are associated with each button in the row, allowing for real-time control of the volume of each individual drum sound sequence.

- **RY30**: Fig. 3.1 c) indicates the instrument pads' area. A pad in this context is an elastic button made of rubber, typically larger than regular buttons. Pads require less precision from, and thus allow greater freedom to, the user – basically allowing the user to "whack" them, without too great of an effort in being spatially precise; approximating the feeling a drummer might have with respectively larger drum surfaces. A step sequencer is accessible by other means (left/right arrow buttons), and likewise for the individual sounds volume control. The pads are able to capture the applied force during the press as a velocity parameter. Additionally, it features a rotary control wheel, which is very convenient for either fast or precise entry of data, unlike a rotary knob (which in small editions can be difficult to set precisely manually).

- **MPC2000**: Fig. 3.1 d) indicates the drum pads' area; due to the machine's nature as a sound sampler, these pads are freely assignable to any sound. Just like with the RY30, a step sequencer is accessible by other means (left/right arrow buttons), and likewise for the individual sounds volume control; its pads are also able to capture the applied force during the press as a velocity parameter.

All of these rhythm machines have facilities in common, that can be (and have been) used in context of live performance, and are achieved by pressing a button, or a chorded sequence of buttons (akin to pressing `SHIFT` along with a different key on a text keyboard) - such as: starting and stopping a sequence, transitioning from one pattern to another, or transitioning from one song to another. Some other common facilities are setting of the master volume, or

setting of the tempo, for which typically (but not necessarily) rotary knob type of controllers might be allocated. However, the unique facilities, otherwise desirable on a generic sequencer interface, might be identified as:

- A button row step sequence controller (as on a TR-808),
- A choice of a "note lane" (the step sequencer applies to) through a rotary control wheel (as on a RY30),
- Individual drum sound "note lane" volumes controllable through a set of mixer faders (as on a SP 1200),
- Drum pads for real-time playback and sequencing (as on an MPC2000, RY30).

This would encompass a basic set of facilities, that are unique to the classic drum machines, but also generically usable in terms of live performance.

Part II

Papers on open soundcard development

Paper A

Extending the soundcard for use with generic DC sensors

Smilen Dimitrov

Refereed article published in the
Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2010), pp. 303–308, Sydney, Australia, June 2010.

The layout has been revised.

Abstract

The sound card anno 2010, is an ubiquitous part of almost any personal computing system; what was once considered a high-end, CD-quality audio fidelity, is today found in most common sound cards. The increased presence of multichannel devices, along with the high sampling frequency, makes the sound card desirable as a generic interface for acquisition of analog signals in prototyping of sensor-based music interfaces. However, due to the need for coupling capacitors at a sound card's inputs and outputs, the use as a generic signal interface of a sound card is limited to signals not carrying information in a constant DC component. Through a revisit of a card design for the (now defunct) ISA bus, this paper proposes use of analog gates for bypassing the DC filtering input sections, controllable from software - thereby allowing for arbitrary choice by the user, if a soundcard input channel is to be used as a generic analog-to-digital sensor interface. Issues regarding use of obsolete technology and educational aspects are discussed as well.

Keywords: Soundcard, Sensors, ISA, DC

A.1 Introduction

The "humble" beginnings of the soundcard[1] as a dedicated part of a PC system intended to produce audible sound, could be seen in the use of a timer circuit Intel 8253, to generate pulses in the audible frequency range and drive a speaker, thereby producing audible sound [1]. Since then, the PC soundcard has become a multichannel A/D interfacing device, able to work at CD quality (16 bits / 44.1 kHz) rates and above. Devices offering more than two output channels are commonly used to drive multiple speakers as part of surround sound home entertainment systems. Specialized soundcards with multiple inputs and outputs, and full duplex (playback while recording) capabilities, have found use in music recording in professional and home studios. Soundcards interface as add-ins to PCs through several busses, serial (USB, FireWire) or parallel (ISA, PCI)[2] in nature; although they are increasingly found integrated in PC motherboards.

A slightly different type of devices become increasingly popular with the academic and do-it-yourself community as A/D interfaces for utilization of sensor signals. Programmable, micro-controller based devices such as the open-source Arduino [2], that offer both A/D conversion and PC connectivity (through, for instance, USB), provide a relatively easy way to interface with a variety of off-the-shelf sensors, from popular software development environments such as

[1] Ignoring earlier occurrences, such as the SID sound chip on a Commodore 64 (whose sound was provided as part of a TV output signal) and similar

[2] USB: Universal Serial Bus; ISA: Industry Standard Architecture; PCI: Peripheral Component Interconnect

`PD` [3], `Max/MSP` [4], or `Processing` [5]. However, these devices are also more limited in regard to sampling quality: for instance, the ARDUINO offers 6 multiplexed channels of 10 bit resolution, and the maximum serial transfer speed via USB is limited to 115200 bps - which puts a theoretical best-case upper limit of 1800 Hz[3] on the sampling rate for all channels.

Because of these limitations, a lot of prototypers and designers opt for a sound-card as a sensor A/D interface instead. This also relieves the designer of worrying about specifics of low level communication, and up-sampling the signals so they match the audio domain processing rate, when sensor signals are to be applied to audio in software. However, since a typical soundcard filters out frequencies outside of the audible 20Hz - 20kHz range, both on the input and the output, its use is limited with those sensing devices that produce output in this range. A soundcard has been in use by Leroy et al for capturing optical pickups [7], or as physical computing interface in context of artwork production [8]; but its use can go beyond musical applications - such as chemical analysis [9] or medicine [10].

The DC[4] bias filtering problem is most apparent with sensors that encode some useful information in the DC level[5]. A common way to circumvent this limitation is to use the DC signal to modulate a sinusoidal carrier in the audio range (usually using amplitude modulation); capture the modulated signal using a soundcard; and then demodulate in software [11]. In case of resistive voltage dividers, they can be driven directly by an AC[4] signal (conveniently, a sound-card can generate a sinusoidal signal for the purpose), in which case the output will conform to the soundcard input limitations. Arguably, there would be some loss of information when using this method - especially if the modulating signal has a spectrum extending above half the the carrier frequency; also, software resources are spent in demodulating a high-speed audio signal, in addition to applying the demodulated signal as a control signal in the application.

On the other hand, direct modifications to commercial sound cards - intended to bypass input sections and allow sampling of DC signals - have also been proposed [12]. In similar fashion, this paper proposes that by using software-controlled analog gates, filtering sections at soundcard inputs can be bypassed - thereby allowing for user-configurable possibility of designating chosen inputs as "sensor" inputs. This relatively simple architectural change, would allow for both high-fidelity acquisition of DC signals, and reuse of common software tools made to interface with soundcards. To test this assumption, a vintage ISA design has been implemented, along with a corresponding C program - discussed further on in this paper.

[3] "*For communicating with the computer, use one of these rates: 300, 1200, ... or 115200.[6]*" Given a 64 bit frame is used to transfer analog data of six channels @ 10 bits, we have best-case period (ignoring start/stop bits) $T = 64[b]/115200[bps] \approx 555\mu s$; and frequency $f = 1/T \approx 1801.8$ Hz; a single 8-bit channel would transfer at 14.4KHz

[4] DC: Direct current; AC: Alternating current

[5] such as a force-sensitive voltage divider, which would provide pressure (or an accelerometer, which would show the influence of gravity) as change of DC level

Therefore, this project aims to demonstrate a simple implementation of PC controlled bypass switches in a soundcard (as an extension to allow interfacing with generic DC sensors), by first implementing and documenting a hardware platform – that can, to some extent, be considered a soundcard.

A.1.1 Approach

The DC bias filtering problem outlined in the Introduction is, in essence, a problem of A/D data acquisition – and thus, an engineering problem – however, one that, arguably, influences a lot of research within electronic music instruments. While custom and suitable A/D hardware may be available on the market, it is often expensive – the cost not scaling with the budget of departments that deal with electronic music. Current affordable A/D hardware (such as Arduino), on the other hand, rarely provides audio-quality sampling rates. The technical specs (audio sampling rate) and ubiquity (low price) of a typical soundcard, then, would make it an ideal "middle-way" choice of A/D platform for electronic music instrument research.

Thus, even though we are talking of, in principle, a relatively simple engineering problem - it is difficult to demonstrate a simple (first-iteration) solution to it, as it is difficult to find an accessible platform *to implement* the solution *on*: this platform needs to behave sufficiently as a soundcard (can interface to a PC, and can perform A/D and D/A conversions at audio rates), and needs to allow space for hardware modifications which is not prohibitively costly.

However, a typical soundcard is an industrial product, aiming to turn a profit by satisfying a range of mass-market needs - and as such, electronic music instrument researchers are unlikely to influence industrial-level modifications to a soundcard product, useful specifically for them. Many soundcards today are single ICs integrated on a PC motherboard, making manual hardware modifications near impossible; and while standalone soundcards may still offer designs based on individual dedicated ICs (allowing more space for tinkering), they will also incur not only greater financial cost, but also a cost involved with understanding the low-level work-flow of the device (which will, most likely, have to be obtained through reverse-engineering, as soundcard manufacturers are unlikely to publish such details publicly).

In such a market environment, researchers are likely to start thinking about handicraft custom-made soundcard implementations. As the typical entry-level research electronics lab, can be likened to a lab accessible for the enthusiast electronics instrument maker, the handicraft approach also shows a promise of direct applicability of results outside of academic circles. Unfortunately, at the end of the (first decade of 20)00's, it is rather difficult to find a starting point for such handicraft development: there are no open public projects dedicated to hardware soundcard implementations, and while there are resources discussing different aspects relevant to development, their discussion level often requires more advanced engineering experience.

In documenting the development process of a soundcard-like hardware, this

project could then also be used in further open discussions of handicraft implementations of soundcard and AD/DA hardware. The starting point for this hardware platform is the only easily readable resource at the time of development, [13], which discusses both hardware and software entry-level issues. Although this design, based on the ISA bus, is old and in market terms obsolete, it has the benefit of using discrete ICs for A/D and D/A converters, as well as for logic signalling. This is, arguably, closer in principle to engineering textbook material, and thus has the educational benefit of facilitating easier understanding of architectural issues surrounding soundcard-like hardware. Additionally, dealing with obsolete technology provides a historical archiving aspect to the project.

This project provides a preliminary general conclusion on the suitability of use of analog switches for interfacing DC sensors, by providing measurements of a generic input signal with a DC component from a signal generator. For educational purposes, the key issues (among them, quantifying the sampling rate of the device) in the process of obtaining these measurements will be outlined in this paper. For more details, as well as source code, consult the webpage [14] - which contains an extended version of this paper, with additional introductory material and implementation details.

A.2 Problem outline

A simplified input channel section of a sound card is shown in Fig. A.1:

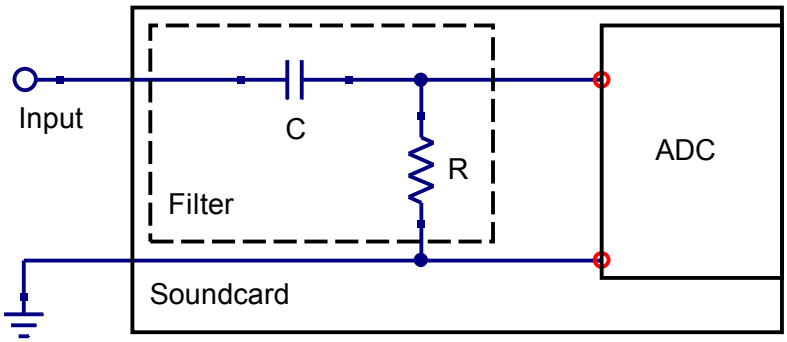


Fig. A.1: Simplified diagram of a sound-card input channel.

In Fig.A.1, a basic CR (*capacitor-resistor*) high-pass filter schematic is taken to represent a simplification of input filters usually found in soundcards, in order to emphasize the filtering of DC signals. To illustrate this influence, a simple experiment can be performed: a stereo mini TRS (*tip-ring-sleeve*) connector can be plugged in a microphone (or line-in) input of a soundcard, and the ground and a channel wire can be used to connect to a 1.5V battery. If we

try to capture the resulting input data using audio recording software (such as `Audacity` [15]), we would obtain a capture like the one shown on Fig. A.2.

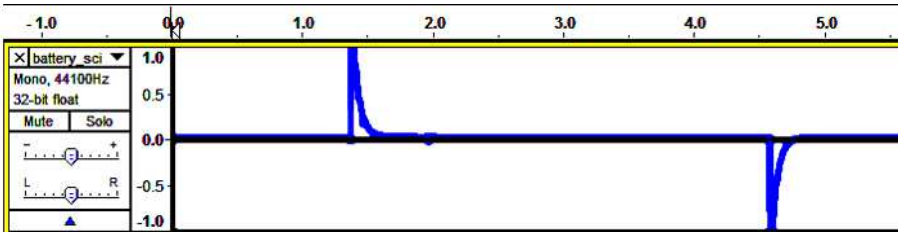


Fig. A.2: Capture of a battery being connected (at 1.5s) and disconnected (at 4.5s) to a sound card microphone input in Audacity software.

Instead of obtaining a constant (DC) level in the time period between approx 1.3s and 4.5s on Fig. A.2, what is shown is a typical signature of a high-pass filter in the time domain - a positive spike at the moment when the battery is connected, and a negative one when it is disconnected. This paper proposes that obtaining DC signals could be achieved by implementing a voltage controlled switch (analog a.k.a. bilateral switches [gates]), ultimately controlled by software, that bypasses the entire input filter section, as shown on Fig. A.3:

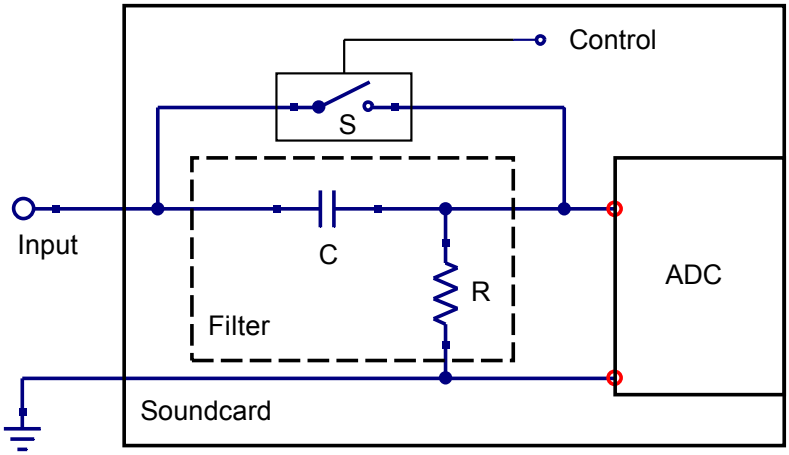


Fig. A.3: Simplified diagram of a sound-card input channel, with a controllable switch bypassing the input filter.

A.3 Soundcard platform

In order to test the assumption given in the problem outline, a hardware platform needs to be chosen, that behaves essentially like a sound-card - but also

allows for demonstration of activating added analog gates via software. Primarily because reverse-engineering an off-the-shelf soundcard device to behave as imagined would have been problematic, but also due to potential educational benefits - a do-it-yourself implementation of a soundcard design was sought instead.

Although the Internet does offer some information and tutorials on building basic extension cards for a PC - interfaced through either USB, ISA or PCI buses - it is difficult to find an available design, which is specifically intended to represent a sound card (or even a high-speed A/D converter). The only one found appropriate for the purpose, is the design for a ISA extension card by Joe D. Reeder [13] - a now abandoned product, that was intended for learning the essentials of software-controlled hardware. Since the schematic and the basic software code, related to this card, are still available on the website associated with this product (www.learn-c.com), it was this design that was taken as a starting point for development.

The schematic for this "Learn-C" ISA card was reimplemented as a double-sided printed circuit board (PCB), with an added **CD4066** [16] CMOS quad bilateral switch. Since the ISA bus is now obsolete and cannot be found in modern PC systems, an older PC based on an Elitegroup ECS P6VAP-A+ (or P6VAP-AP) motherboard, with a single ISA slot, was obtained. Code in C language from [13] was used to implement test software, and Agilent 54621a oscilloscope was used to capture signals on board - using the open-source `agiload` [17] package for transferring oscilloscope traces to a PC as raw data. The experiment finally consisted of using a vintage Wavetek model 145 signal generator to produce an approx. 440 Hz sinusoid voltage with a DC offset level, and capturing this voltage with the "Learn-C" card and test software on disk. As the test software allowed for user-controlled activation of the bilateral switches, these were alternately turned on and off during capture - and the captured signal was observed in the open-source `Audacity` software.

A.3.1 ISA hardware implementation

The original schematics for the "Learn-C" ISA card found in [13] was rebuilt using open-source `KiCad` [18] software (Fig. A.4); the same software was also used to produce the PCB layout. The design relies only on the ISA bus power supply pins 1, 3, 5, 7, 9, 10, 29, 31 (*GND, +5V, -5V, -12V, +12V, GND, VCC, GND*) as well as ISA IO pins 2, 13, 14, 20, 33-40, 42, 53-62 (*RESET, IOW, IOR, OSC, D0-D7, AEN, A9-A0*), for power supply and PC connectivity.

In simple terms, whenever a I/O command like `outportb(_port, _data )` is executed from software, a binary representation of the `_port` address is set up as respective high or low voltages on the address pins of the ISA bus; the "Learn-C" design then employs standard **74xx** TTL logic ICs to implement an address decoder that will interface with the bus, and provide appropriate trigger signals for the rest of the hardware, when the card is addressed from software.

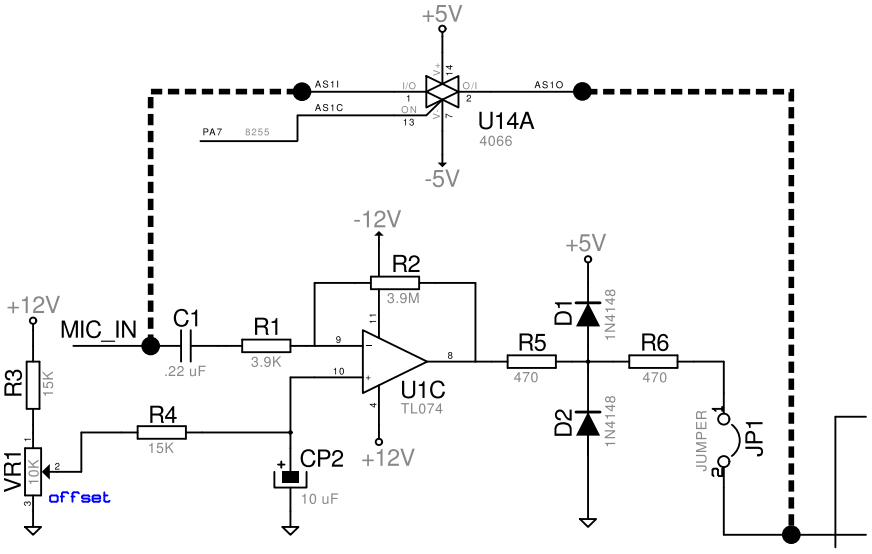


Fig. A.4: Part of schematic of the ISA card; emphasized dashed lines indicate connections of bipolar switch (see [14] for a complete schematic).

Most of the original design of the "Learn-C" ISA card has been reproduced, although with some differences. For instance, a socket for the **CD4066** analog switch was added, and not all wired connections were implemented on the PCB (for instance, headers were left unconnected, as well as most of the I/O port pins of the **8255** [19] PPI chip). On the other hand, both **DAC0832** [20] digital-to-analog converters (DAC) were implemented[6]. The implementation of the card is shown on Fig. A.5.

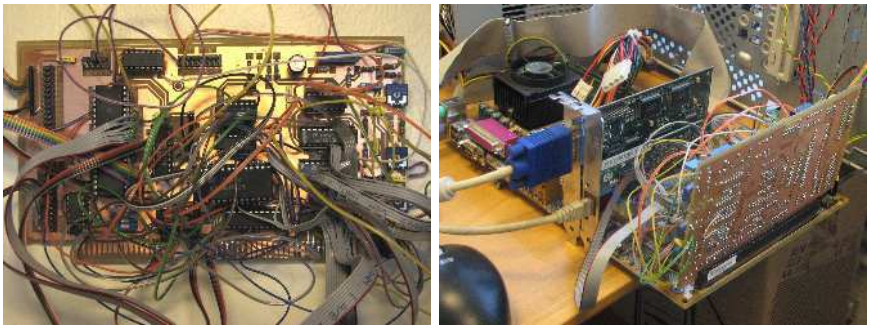


Fig. A.5: Left: image of the wired card; right: card in ISA slot of motherboard of test PC.

[6]although only a single one was actually tested; and none are needed for the input filter switching experiment.

A.3.2 Software

As mentioned previously, the source code for the "Learn-C" ISA card given in [13] is C code, originally intended to run under `MS-DOS`; as part of this project, portions of that code were ported to `Linux` as well. The "Learn-C" example programs can also run in the command prompt shell of `Windows XP` - however, they cannot be directly compiled with modern `Windows` C compilers. The reason for this is that the code relies on C commands like `inportb` and `outportb` (or `inp` and `outp`), which represent direct I/O port access; however direct I/O port access is disabled for `Windows` architectures newer than `NT` [21] (and can be achieved only through programming a device driver). This demanded use of vintage C compilers (`DJGPP` for `DOS`) under `Windows`; on the other hand, programming direct I/O port access in C under `Linux` is relatively straightforward.

A.4 Testing procedure

The testing procedure consisted of two distinct steps. The first was to use a known signal to determine the sampling rate of the analog-to-digital converter (ADC); and to check whether this rate is correct (by auditorily comparing a capture from the ISA card's ADC to the known signal). The second step (the actual experiment) consisted of sampling and capturing a known, DC-biased, sinusoidal signal; turning the analog switch on and off during capture; and looking for presence of a DC level recording when the switch was active in observations of captured data.

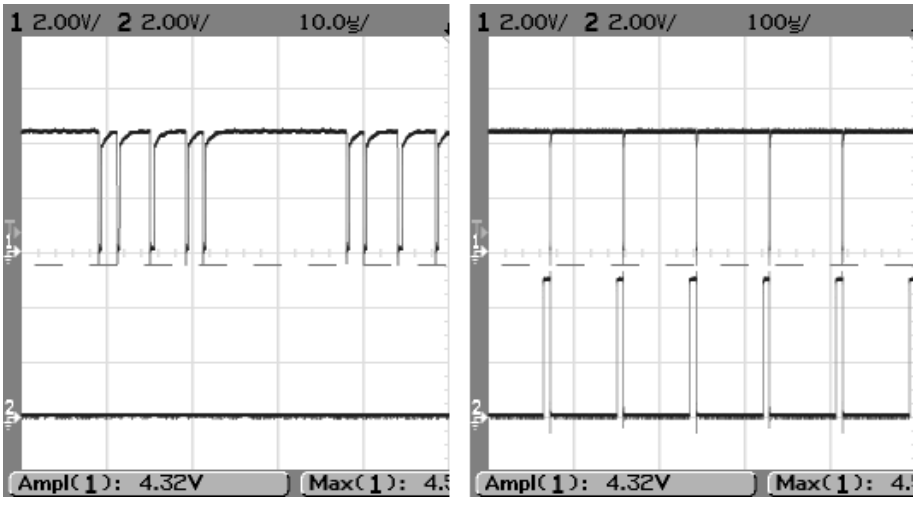


Fig. A.6: Oscilloscope screen captures of card signals `EOC` (top) vs `IOW` (bottom). Left: 50 μs of non-periodic behavior; right: 0.5 ms of periodic behaviour.

A.4.1 Determining the ISA card sampling rate

The ISA card sampling rate was determined to be around 12725 Hz (the sampling resolution is limited at 8 bit by choice of **ADC0809** [22] chip). However, this could be reached only after dealing with an apparent timing problem.

Figure A.6 (left) shows that `EOC` (signal produced by the ADC chip on the card at end of each sample conversion) is not periodic. Eventually, it turns out that this can be resolved by introducing a short delay between the `outport` and the first `inport` command in the reading loop, as shown in Listing A.1:

```
unsigned base; unsigned adcport; base = adcport = 0x200; //1000000000
if ((fp = fopen(FILENAME,"w+b")) != NULL) {
  while(!kbhit())
  {
    outp(adcport, 0); // start channel 0 conversion , by writing whatever value to address
          adcport
    iz = del; while (iz > 0) iz--; // fake delay, ' del ' increments
    while(!(inp(adcport+0x18) & 0x80)); // wait for EOC ready: 0x18 = 000011000, 0x80 =
         010000000
    x = inp(adcport); // read ADC value into variable x
    fputc((char)x,fp); // since value is 8-bit anyways , just cast to char and save to
         disk
  }
}
```

Listing A.1: ADC reading loop code

As Listing A.1 indicates, the recorded file is simply a stream of 8-bit characters. This file can be imported in `Audacity` as raw data, and the sampling frequency is set upon import. The multi-track capabilities of `Audacity` also allow both the original 440 Hz source signal, and its ADC capture from the ISA card, to be played simultaneously in spite of differing sampling rates (44.1 KHz vs. 12.7 KHz); their respective pitches can be heard as audibly close – which is a confirmation that the measurement of the sampling rate is correct.

A.4.2 Test of analog switch functionality

As mentioned previously, a **CD4066** was used to implement an analog switch, which bypasses the input preamp/filter section of the ISA card. This chip offers four analog switches - only a single one was used, defined by pins 1 and 2 as switch connectors (connected as on Fig. A.4), and pin 13 (**CD4066**/p13) as voltage control. To control **CD4066**/p13, the **8255** PPI on the ISA card was used, as it offers three ports (A, B and C) of 8 pins each, which can be configured to act as either digital inputs, or digital outputs. Just a single output pin is needed from a single (configured as output) port, in order to control the analog switch; pin 37 (or pin 7 of port A) of **8255** (**8255**/p37) was picked for the purpose, and was connected to **CD4066**/p13. The **8255** offers three different modes of configuration of its three ports; here any mode that configures port A as output will do, and the function `set_up_ppi` from [13] was used to quickly configure the ports.

A.5 Results

The procedure described in section A.4.1 was used to capture a DC offset sinusoidal signal. This signal was generated by a vintage WAVETEK MODEL 145 signal generator, which doesn't provide for fine-tuning control of the AC amplitude and the DC offset separately. Eventually, a signal spanning between 0.5V and 1.66V, set at approximately 450 Hz, was used as the input signal for ADC capture. The signal arriving at the input pin **ADC0809**/p26 (IN0), without and with the influence of the switch, is shown on Fig. A.7.

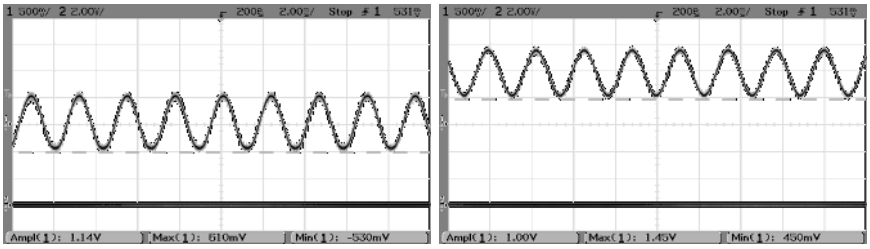


Fig. A.7: Oscilloscope screen captures of the input signal brought to ADC0809 IN0 (left) without the influence of analog switch; (right) while analog switch turned on

This input signal was captured on a disk file using the procedure given in section A.4.1, as the analog switch was turned on and off during capture using keyboard key presses. The data captured by the ISA card can be verified through an import in `Audacity`, depicted in Fig. A.8.

As it is obvious from Fig. A.8, activating the switch allows the DC level of the signal to be captured by the card's ADC; and the waveforms obtained in `Audacity` remain relatively faithful to the original input signal shown in Fig. A.7 (ignoring the clipping of the negative semiperiod of the original signal).

A.6 Discussion

As we have shown that, with this platform, we can arbitrarily choose to capture the DC level of a signal from a signal generator – we can conclude that the same behavior can be expected for signals derived from sensor circuits (for instance, a FSR-based voltage divider) as well. Thus, this paper indicates that analog switches bypassing the input preamplifier section of soundcard inputs, would be a relatively simple change to perform on existing soundcard designs, thereby expanding their purpose to high speed A/D interfaces for generic sensors. Similar change could be implemented for output sections, thereby allowing soundcards to be used as generic signal generators. The DC blocking capacitors are present in a sound-card, of course, for a purpose - primarily to protect speakers from constant DC biasing, and thereby prolong their lifetime [23]. The design change proposed here, would ideally leave that regime unchanged for audio purposes -

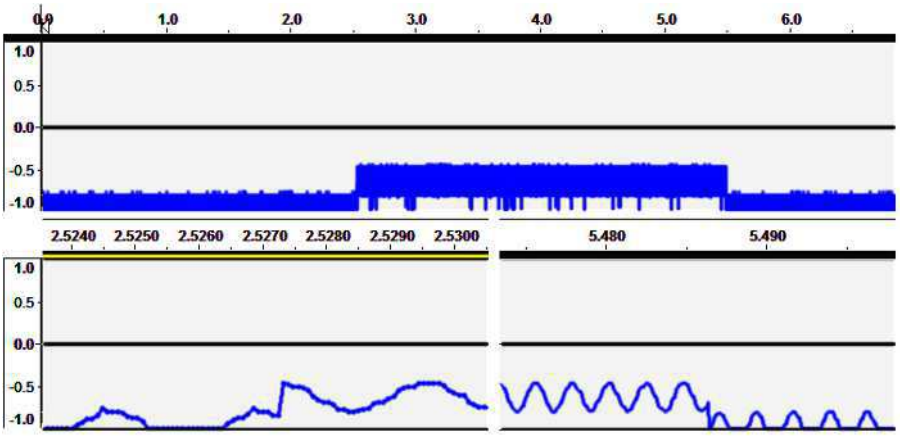


Fig. A.8: Top: Six second capture of an input signal with DC, with the switch activated in mid-capture; Bottom: temporal zoom at moments of activating (left) and deactivating (right) the switch.

and simply introduce a different one, more suitable for sensor data acquisition, when the user requires it.

A.6.1 The soundcard platform

This project started by looking for a hardware platform, that behaves essentially like a soundcard. Arguably, the ISA platform used here cannot be even considered a sound card, until it can be used by typical audio software (like players or editors) from the application level in an operating system. Although this ISA card could be extended to play back audio (encoded for the measured DAC frequency) - at the state presented here, it behaves more like a generic signal generator and sampler, than a modern soundcard. Finding a modern open hardware platform, that allows for the type of research as in this project, is still problematic; although FPGA[7] based designs, such as the ones described in [24], are very promising as a base for multichannel, high-speed, hybrid audio/sensor interfaces.

Whereas it is utopistic to expect that this paper could significantly influence industry in such a manner, that similar modifications become a standard for future sound-cards - it certainly aims to inspire designers working in the electronic music instrument field, to focus at the intricacies of digital interfacing with analog signals; and to consider using older, historic designs for appropriate purposes - while being aware of potential obstacles. Mostly, one has to deal with hardware availability, although software can be an issue as well. However, in the case of ISA, this project shows that currently there is still a palette of tools that can target such machines and corresponding functionality, many of

[7]FPGA: Field-programmable gate array

them free and open-source; yet, one has to be prepared for a time investment for straightening out potential glitches.

Hardware Arguably, the ISA design used as a test platform here, doesn't even come close to issues in contemporary sound-card production – to begin with, negative voltage values of the input signal are clipped, as the ADC chip by default can sample only positive voltages. However, it is a good educational tool to introduce general issues related to design of soundcards and corresponding software. Namely, it is often difficult for beginner engineers to come to a practical example, which is both relatively simple to understand as introductory material (and thus easy to relate to theory) - and can be practically implemented to serve a purpose, already known from a user perspective. So in spite of the obsolescence of ISA, this design can still be useful educationally. One positive point of using discrete components in this card implementation, is the possibility to measure their signals individually with an oscilloscope and thus observe the interdependence of different signals on the physical, electric level - something that becomes arguably difficult, if the components are integrated as part of a single chip.

Software Finally, if there is a single thing this project points out, it is that one cannot perceive the soundcard as a system separate from its host PC; that is, a soundcard-like hardware is genuinely bound to the intricacies of the host operating system. In other words, even if the hardware is theoretically capable of achieving greater sampling speeds, the effective achieved sampling rate will be limited by the method of accessing the device from software. In this project, a steady 12.7 KHz sampling rate was achieved through a program in C, only after implementing delays in the reading loop. The cause of this need for delay is, likely, that `outportb` in Listing A.1 returns, before the corresponding `IOW` signal (which triggers each start of conversion) has been set electrically. Hence we need to wait, so we're sure conversion has started – before reading the `EOC` signal to see if a value is ready. However, due to operating system overhead, the actual sampling period is probably more indicative of the time it takes for the operating system to complete a single iteration of the main `while` loop in Listing A.1, than it is of the limits of the hardware itself. Thus, entering development of soundcard hardware, necessarily implies involvement with issues in the inherent unpredictability of instruction execution times, OS scheduler granularity and latency [25] and development of device drivers [27], [26] – with a particular focus on the flow of time [28] – before coming close to a platform able to interface with an operating system, in a manner expected of a soundcard. In other words, here the hardware acquisition of *each* sample is initiated and transferred by C software on the PC – with a proper driver approach, the hardware acquisition process could run independently on the card hardware at high rates; and batches of sampled data could be transferred as arrays to the PC at longer intervals (i.e. "*buffered reads*"; see also Direct Memory Access/DMA [29]). Note, however, that in spite of the shortcomings of

this projects' naïve approach, it still managed to obtain 12.7 KHz effective end-to-end sampling rate, which is close to the 14.4 KHz USB-limited maximum for a single channel, 8-bit Arduino transfer (but is still below the CD-quality 44.1 KHz expected of a typical soundcard).

A.7 Conclusion

In conclusion, the project managed to demonstrate the possibility to use analog switches, for software controlled bypass of input filters of a sound-card device; thereby, in principle, allowing it to interface with generic sensor circuits that produce DC-offset voltages. However, claims cannot be made on the feasibility of implementing such a change in an existing commercial sound-card design. The project also illustrated the specific problems encountered with usage of a card design for the now obsolete ISA bus; while demonstrating how it can be used to emulate modern hardware (at least to a degree, sufficient to expose the problem at hand). Additionally, the simplified analysis of particular issues, aims to serve as an educational introductory example for designers starting with digital hardware design; in line with this aspect, the source files for schematics and code, as well as the full list of online references (too numerous to include here) relevant to the topic discussed in this paper, are provided on the project webpage [14]. The educational and the historical perspective of this project, both aim to contribute to furthering the discussion of A/D and D/A hardware in the context of electronic music instruments development.

References

[1] T.C. Savell, "23.3 Digital Audio Processors for Personal Computer Systems", *Linear Algebra and Ordinary Differential Equations*, 1993.

[2] arduino.cc, "Arduino homepage", `http://www.arduino.cc/`, web page.

[3] puredata.info, "Puredata homepage", web page. URL: `http://puredata.info/`.

[4] Cycling 74, "Max/MSP homepage", web page. URL: `http://cycling74.com/products/maxmspjitter/`.

[5] processing.org, "Processing homepage", web page. URL: `http://processing.org/`.

[6] Arduino, "Arduino - Begin", web page, 2010. URL: `http://www.arduino.cc/en/Serial/Begin`.

[7] N. Leroy, E. Fléty, and F. Bevilacqua, "Reflective optical pickup for violin", in *Proceedings of the 2006 conference on New interfaces for musical expression*, IRCAM-Centre Pompidou Paris, France, France, 2006, pp. 204–207.

[8] Kazuhiro Jo, "Audio Interface as a Device for Physical Computing", *Proceedings of Audio Mostly 2008 - a Conference on Interaction with Sound*, pp. 123–127, 2008. URL: http://www.jojporg.dreamhosters.com/public_dav/paper/audiomostly08-audiointerface-jo.pdf.

[9] D. Nacapricha, N. Amornthammarong, K. Sereenonchai, P. Anujarawat, and P. Wilairat, "Low cost telemetry with PC sound card for chemical analysis applications", *Talanta*, vol. 71, no. 2, pp. 605–609, 2007.

[10] K.A. Reddy, J.R. Bai, B. George, N.M. Mohan, and V.J. Kumar, "Virtual Instrument for the Measurement of Haemo-dynamic Parameters Using Photoplethysmograph", *Proc 23rd Int ConfIEEE, IMTC-2006*, pp. 1167–1171, 2006.

[11] M.J. H. Puts, J. Pokorny, J. Quinlan, and L. Glennie, "Audiophile hardware in vision science; the soundcard as a digital to analog converter", *Journal of Neuroscience Methods*, vol. 142, no. 1, pp. 77–81, 2005.

[12] Scott Molloy, "How to Modify a PC Sound Card to Allow D.C. Voltage Measurements", web page, Last Accessed: 7 April, 2009. URL: http://web.archive.org/web/20080108175023/http://www.mandanet.net/adc/adc.shtml.

[13] Joe D. Reeder, "Tutorial - Controlling The Real World With Computers", web page, 2005. URL: http://learn-c.com/ (visited on 03/12/2009).

[14] Smilen Dimitrov, "Extending ISA Soundcard webpage", web page, Last Accessed: 20 March, 2009. URL: http://imi.aau.dk/~sd/phd/index.php?title=ExtendingISASoundcard.

[15] audacity.sourceforge.net, "Audacity homepage", web page. URL: http://audacity.sourceforge.net/.

[16] Fairchild Semiconductor, *CD4066 datasheet*, 2005. URL: http://www.fairchildsemi.com/ds/CD/CD4066BC.pdf.

[17] Jürgen Rinas, "agiload - fetch data and screenshots from Agilent 5462x oscilloscopes", web page. URL: http://www.ant.uni-bremen.de/whomes/rinas/agiload/.

[18] Jean-Pierre Charras, "KiCad homepage", web page, 2010. URL: http://www.lis.inpg.fr/realise_au_lis/kicad/.

[19] Intel Corporation, *8255 Programmable Peripheral Interface (PPI)*, 1995. URL: http://jap.hu/electronic/8255.pdf.

[20] National Semiconductor, *DAC0830/DAC0832 datasheet*, 2002. URL: http://www.national.com/ds/DA/DAC0830.pdf.

[21] Dale Roberts, "Dr. Dobb's - Direct Port I/O and Windows NT", web page, Last Accessed: 12 April, 2009. URL: http://www.ddj.com/184409876?pgno=3.

[22] National Semiconductor, *ADC0808/ADC0809 datasheet*, 2009. URL: http://www.national.com/ds/DC/ADC0808.pdf.

[23] www.maxim-ic.com, "APPLICATION NOTE 3979 - Overview of DirectDrive® Technology", web page, Last Accessed: 12 April, 2009. URL: http://www.maxim-ic.com/appnotes.cfm/an_pk/3979/.

[24] S. Kartadinata, "The Gluion advantages of an FPGA-based sensor interface", in *Proceedings of the 2006 conference on New interfaces for musical expression*, IRCAM-Centre Pompidou Paris, France, France, 2006, pp. 93–96. URL: http://www.glui.de/gluion/gluion.pdf.

[25] Theodore P. Baker, An-i Andy Wang, and Mark J. Stanovich, "Fitting linux device drivers into an analyzable scheduling framework", in *Proceedings of the 3rd Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2007.

[26] C. Williams, "Linux scheduler latency", *Red Hat Inc*, vol. 3, 2002.

[27] Michael K. Johnson, "Device Driver Basics", web page, 2010. URL: http://users.evtek.fi/~tk/rt_html/ASICS.HTM.

[28] Alessandro Rubini and Jonathan Corbet, "Flow of Time", in *Linux device drivers*, Second, O'Reilly Media, Inc., 2001, ch. 6.

[29] ——, "mmap and DMA", in *Linux device drivers*, Second, O'Reilly Media, Inc., 2001, ch. 13.

Paper B

Minivosc - a minimal virtual oscillator driver for ALSA (Advanced Linux Sound Architecture)

Smilen Dimitrov and Stefania Serafin

Refereed article published in the
Proceedings of the Linux Audio Conference (LAC 2012), pp. 175–182,
Stanford, California, USA, April 2012.

The layout has been revised.

Abstract

Understanding the construction and implementation of sound cards (as examples of digital audio hardware) can be a demanding task, requiring insight into both hardware and software issues. An important step towards this goal, is the understanding of audio drivers - and how they fit in the flow of execution of software instructions of the entire operating system.

The contribution of this project is in providing sample open-source code, and an online tutorial [1] for a mono, capture-only, audio driver - which is completely virtual; and as such, does not require any soundcard hardware. Thus, it may represent the simplest form of an audio driver under ***ALSA****, available for introductory study; which can hopefully assist with a gradual, systematic understanding of* ***ALSA*** *drivers' architecture - and audio drivers in general.*

Keywords: Sound card, audio, driver, ALSA, Linux

B.1 Introduction

Prospective students of digital audio hardware, could choose the sound card as a topic of study: on one hand, it has a clear, singular task of managing the PC's analog interface for playback and capture of digital audio data - as well as well-established expectations by consumer users in terms of its role; on the other hand, its understanding can be said to be cross-disciplinary, as it encompasses several (not necessarily overlapping) areas of design: analog and digital electronics related to soundcard hardware and PC bus interface implementation; PC operating system drivers; and high-level PC audio software.

Gaining a sufficient understanding of the interplay between these different domains in a working implementation can be an overwhelming task; thus, not surprisingly, the area of digital audio hardware design and implementation (including soundcards) is currently dominated by industry. Recent developments in open source software and hardware may lower the bar for entry of newcomer DIY enthusiast - however, the existence of many open source drivers for commercial cards doesn't necessarily ease the introductory study of a potential student.

In essence, an implementation of a soundcard will eventually demand dealing with the issue of an operating system *driver*. In the current situation, a prospective student is then faced with a 'chicken-and-egg' problem: proper understanding of drivers requires knowledge of the hardware (which the drivers were written for); and yet understanding the hardware, involves understanding of how the drivers are supposed to interface with it[1]. A straightforward way out, would be to study a 'virtual' driver - that is, a driver not related to an actual hardware; in that case, a student would be able to focus solely on the

[1] and the lack of open card hardware designs for study makes this problem more difficult

software interaction between the driver, and the high-level audio program that calls it. Unfortunately, in the case of the **ALSA** driver architecture for **Linux**, pre-existing examples of virtual drivers are in fact not trivial[2] - and, just as existing **ALSA** driver tutorials, *assume previous knowledge* of bus interfaces (and thus hardware).

The **minivosc** driver source code with the corresponding tutorial (on the **ALSA** website [1]) represents the simplest possible virtual **ALSA** driver, that does not require additional hardware. It has already led to the development of the driver used in the (possibly first) demonstration of an open soundcard system in **AudioArduino** [3a] (and further used in [5a]) - and as it limits the discussion to *only* the *software* interaction between driver and high-level software, disregarding issues in bus interfacing and hardware - it would represent a conceptually simpler entry level for a prospective student of sound card drivers.

B.2 Premise

Personal computer users working with audio typically rely on high-level *audio software* (from media players such as **VLC**, to more specialized software like **Pure Data**, or the wave editor **Audacity**[3]) to perform their needed tasks – and the *sound card* (as hardware) to provide an analog interface to and from audio equipment. This necessarily puts demands on the operating system of the PC, to provide a standardized way to access (what could be different types of) audio hardware. An operating system, in turn, would provide an audio or soundcard *driver* API (application programming interface), which should allow for programming of a driver that: abstracts some of the 'inner details' of the soundcard implementation; and exposes a standardized interface to the high-level audio software (that may want to utilize this driver). This, in principle, allows interfacing between software and hardware released by different vendors/publishers.

Earlier work like [1a] attempts to provide a systematic approach to soundcard implementation; however, one clear conclusion from such a naïve approach is that: regardless of the capabilities of the hardware - one cannot achieve a fine control of timing required for audio, by using what corresponds to a simple 'user space' C program. Problems like these are typically solved within the driver programming framework of a given operating system - and as such, acquaintance with driver programming becomes a necessity for anyone aiming to understand development of digital audio hardware for personal computers. In terms of FLOSS[4] **GNU/Linux**- based operating systems, the current driver

[2] and may require existence of real soundcards on the system

[3] Note that software like JACK - while it can be considered more 'low-level' than consumer audio software - is still intended to route data between 'devices'. Since it is the *driver* that provides this 'device' (as a OS construct that software can interface with) in the first place - drivers lay in a lower architectural layer than even software like JACK, and so involve different development considerations.

[4] free/libre/open source software

programming framework - as it relates to soundcards and audio - is provided by the Advanced Linux Sound Architecture (**ALSA**). **ALSA** supersedes the previous OSS (Open Sound System) as the default audio/soundcard driver framework for **Linux** (since version 2.6 of the kernel [2]), and it is the focus of this paper, and the eponymous **minivosc** driver (and tutorial). The **minivosc** driver was developed on **Ubuntu** 10.04 (Lucid), utilizing the 2.6.32 version of the **Linux** kernel; the code has been released as open source on Sourceforge, and it can be found by referring to the tutorial page [1].

B.2.1 Initial project issues

The **minivosc** project starts from the few readily available (and 'human-readable') resources related to introductory **ALSA** driver development: [3], [4], [5], and [6]. Most of these resources base their discussions on conceptual or undisclosed hardware, making them difficult to read for novices. On the other hand, there are few examples of virtual soundcard drivers, such as the driver source files **dummy.c** (in the **Linux** kernel source tree [7]) and **aloop-kernel.c** (in the **ALSA** source tree [8]); however, these drivers don't have much documentation, and can present a challenge for novices[5]. All these resources [5], [3], [4], [6]–[8] have been used as a basis here, to develop an example of a **mini**mal **v**irtual **osc**illator (**minivosc**) driver.

B.3 Architectural overview of PC audio

Even if the **minivosc** driver is a virtual one, one still needs an overview of the corresponding hardware architecture - also for understanding in what sense is this driver 'virtual'. As a simplified illustration, consider Fig. B.1.

A driver will typically control transfers of data between the soundcard and the PC, based on instructions from high-level software. The direction from the soundcard to the PC is the *capture* direction; the opposite direction (from the PC to the soundcard) is the *playback* direction; a soundcard capable of delivering both data transfer directions simultaneously can be said to be a *full-duplex* device.

While Fig. B.1 shows the hard disk as (ultimately) both the source for the playback direction, and the destination for the capture direction - within this process, the CPU may use RAM memory at its discretion. In fact, the driver is typically exposed to pointers to byte arrays (buffers) in memory (in **ALSA** known as PCM (sub)streams [9, '*PCM (digital audio) interface*'], and named **dma_area**), that represent streams in each direction.

[5]the 'dummy' driver doesn't actually perform any memory transfers (which is, arguably, a key task for a driver), so it cannot be used as a basis for study – the 'loopback' driver is somewhat more complex than a basic introductory example, as it is intended to redirect streams between devices, and as such assumes some preexisting acquaintance with the **ALSA** architecture

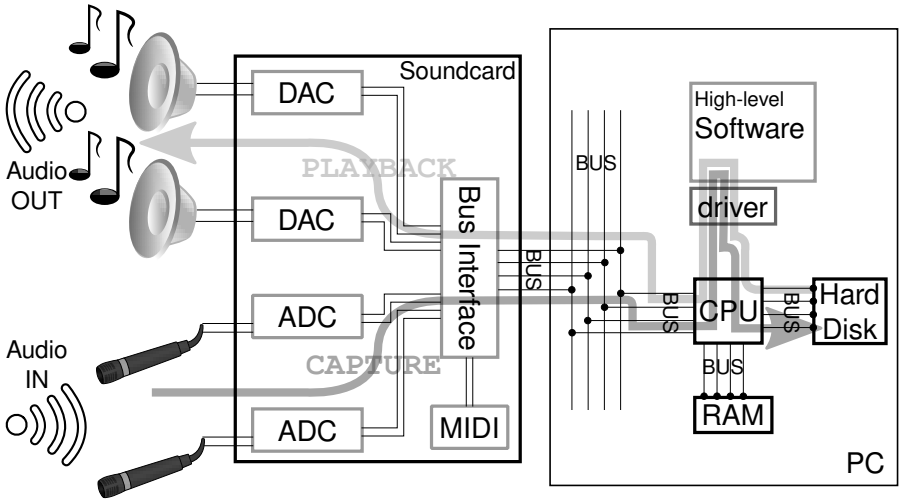


Fig. B.1: Simplified overview of the context of a PC soundcard driver (portions used from Open Clip Art library).

In terms of audio streams, Fig. B.1 demonstrates a device capable of two mono (*or one stereo*) inputs, and two mono (*or one stereo*) outputs. Since audio devices like microphones (or amplifiers for speakers) typically interface through analog electronic signals - this implies that for each 'digital' input [or output] audio stream, a corresponding analog-to-digital (ADC) [or digital-to-analog (DAC)] converter hardware needs to be present on the soundcard[6].

As the main role of the soundcard is to provide an analog electronic audio interface to the PC - the role of the ADC and DAC hardware is, of course, central. However, the PC will typically interface to external hardware through a dedicated bus for this purpose[7]. This means, that some *bus interfacing* electronics - that will decode the signals from the PC, and provide signals that will drive (at the very least) the ADC/DAC converters - needs to be present on the soundcard as well[8].

An **ALSA** driver uses a particular terminology when addressing these architectural surroundings. The 'soundcard' on Fig. B.1 will be considered to be a **card** by the driver[9]. One level deeper, things can get a bit more complicated:

[6]Note, however, that this correspondence could, in principle, be solved by a *single* ADC or DAC element - along with a (de)multiplexer which would implement time-sharing of the element (for multiple channels)

[7]noting that, in principle, the buses used for hard-disks (such as *Integrated Drive Electronics (IDE)*) or RAM (known as *'Memory Interconnect'*) can be distinct

[8]For example, [1a] describes a device that interfaces through the *Industry Standard Architecture (ISA)* bus - and uses standard TTL components (such as **74LS08**, **74LS688**, **74LS244**, etc) to implement a bus interface; [3a] describes a device that interfaces through the *Universal Serial Bus (USB)* - and uses the **FT232** chip by FTDI to implement a bus interface

[9]noting that, in principle, the driver should be able to handle *multiple* cards; and be able

assuming that Fig. B.1 represents a *stereo* soundcard, it would have one input stereo connector (attached to two ADCs), and one output stereo connector (attached to two DACs); an **ALSA** driver would correspondingly be informed about a card, that has one stereo input **device** (consisting of two **subdevice**s) - and one stereo output device (consisting, likewise, of two subdevices). Note that: "*..we use subdevices mainly for hardware which can mix several streams together* [10]" and "*typically, specifying a sound card and device will be sufficient to determine on which connector or set of connectors your audio signal will come out, or from which it is read... Subdevices are the most fine-grained objects ALSA can distinguish. The most frequently encountered cases are that a device has a separate subdevice for each channel or that there is only one subdevice altogether* [11]"

The **ALSA** driver is informed about such a hierarchical relationship (between *card*, *device*s and *subdevice*s) through structures (C `struct`s, written by the driver author in the driver source files) - defined mostly through use of other structures, predefined by the **ALSA** framework (alias the **ALSA** 'middle layer'). The driver code, additionally, establishes a relationship between these structs, and the PCM stream data that will be assigned to each in memory; and connects these to predefined **ALSA** framework driver functions, which define the driver (and the corresponding hardware) behavior at runtime. Finally, Fig. B.1 shows that other types of devices, such as a MIDI interface, can also be present on the soundcard. The **ALSA** framework has facilities to address such needs too - as well as having a so-called *mixer* interface[10] - which will not be discussed here.

Application level From the PC perspective, a high-level audio software (audio application) is used, in first instance, to issue start and stop of audio playback or capture. When such a high-level command is issued by the user, the audio application communicates to the driver through the application-level API and: obtains a handle to the relevant structures; initializes and allocates variables; opens the PCM device; specifies hardware parameters[11] and type of access (interleaved or not) - and then starts with reading from (for capture) or writing to (for playback) the PCM device, by using **ALSA** API functions (such as `snd_pcm_writei`/`snd_pcm_writen` or `snd_pcm_readi`/`snd_pcm_readn`) [12]. The PCM device is representation of a source (or destination) of an audio stream[12]. The kernel responds to the application API calls by calling the respective code in the kernel driver, implemented using the kernel (**ALSA** driver) API [3].[13]

to individually address each one

[10] which allows for, say, individual volume control directly from the main OS volume applet

[11] access type, sample format, sample rate, number of channels, number of periods and period size

[12] and it can have: "plughw" or "hw" interface; playback or capture direction; and standard (blocking), non-blocking and asynchronous modes (see also [9, 'PCM (digital audio) interface'])

[13] Note that the application doesn't have to talk to the driver directly; there could be intermediate layers, forming a **Linux** audio software stack (see [13]). However, in this paper,

B.4 Concept of `minivosc`

A user would, arguably, expect to hear actual reproduced sound upon clicking 'play'; while recording, in principle, doesn't involve user sensations other than indication by the audio software (e.g. rendering of captured audio waveform). Taking this into account, it becomes clear that the stated purpose of **`minivosc`** - to be a 'virtual' driver (independent of any actual additional soundcard hardware) - can only be demonstrated in the *capture* direction[14]: as the driver simply has references to data arrays in memory, the effect of *playing back* (i.e., copying) data *to* non-existing hardware will be pretty much undetectable[15]. However, even with non-existing hardware, we can always write some sort of predefined or random data to the capture buffers in memory - which would result with visible incoming data in the high-level audio software (like when performing 'record' in **`Audacity`**).

To avoid the conceptualization problems of **`ALSA`** devices vs. subdevices, the **`minivosc`** driver is deliberately defined as a mono, 8-bit, capture-only driver, working at 8 kHz (the next-lowest[16] rate ALSA supports). The 8-bit resolution allows also for direct correspondence between: the digital representation of a single analog sample; and the storage unit of the corresponding arrays (buffers) in memory, which are defined as `char*`. Hence, one byte in memory buffer represents one analog sample, the next byte represents the next analog sample, etc. This allows for simplification of the process of wrapping data in a ring buffer, and thus easier grasping of the remaining key issues in **`ALSA`** driver implementation.

B.5 Driver structures

The **`minivosc`** driver contains four key structures - three of which are required by (and based on predefined types in) the **`ALSA`** framework:

- `struct` variable of type `snd_pcm_hardware` (required) - sets the allowed sample formats, sampling rates, number of channels, and buffering properties
- `struct` variable of type `snd_pcm_ops` (required) - assigns the actual functions, that should respond to predefined **`ALSA`** callbacks

we focus solely on the perspective of the **`ALSA`** kernel driver.

[14]however, note that **`aloop-kernel.c`**[8], is also a 'virtual' driver, and yet works in both directions; however, since it is intended to 'loop back' audio data between applications and devices[14], the virtual setups possible can be reduced to the case when the 'loopback' driver routes one audio application's data written to its playback interface, back to its capture interface; and another audio application grabs data from the 'loopback' capture interface and writes it to disk.

[15]similar to, in **`Linux`** parlance, 'piping' data to **`/dev/null`**. While a specific consumer of such data could be programmed, that alone complicates the understanding of interaction between typical audio software and drivers

[16]The lowest **`ALSA`** rate being 5512 Hz, see **`include/sound/pcm.h`** in **`Linux`** source [15]

- `struct` variable of type `platform_driver` (required) - named `minivosc_driver` , it describes the driver, and at the same time, determines the bus interface type
- `struct` variable of type `minivosc_device` - custom structure that contains all other parameters related to the soundcard, as well as pointers to the digital audio (PCM) data in memory

The `minivosc_driver` struct variable defines the `_probe` and `_remove` functions, required for any **Linux** driver; however, by choosing the struct type, we also determine the type of bus this driver is supposed to interface through. For instance, a PCI soundcard driver would be of type `struct pci_driver`; whereas a USB soundcard driver would be of type `struct usb_driver` (see [1]). However, `minivosc` is defined as `platform_driver`, where "*platform devices are devices that typically appear as autonomous entities in the system* [16, 'platform.txt']" - and as such, it will not need actual hardware present on any bus on the PC, in order for the driver to be loaded completely[17].

The `snd_pcm_ops` type variable simply points to the actual functions that are to be executed as the predefined **ALSA** callbacks, which are discussed in the next section. The different fields in `snd_pcm_hardware` allow the device capabilities in terms of sampling resolution (i.e., analog sample format) and sampling rate to be specified. For this purpose, there are predefined bit-masks in **ALSA**'s `pcm.h` [15], such as `SNDRV_PCM_RATE_8000` or `SNDRV_PCM_FMTBIT_U8` (for 8 kHz rate, or for sample format of 8-bit treated as unsigned byte, respectively). One should be aware that audio software may treat these specifications differently: for instance, having **arecord** capture from the **minivosc** driver, will result with an 8-bit, 8 kHz audio file - simply because that is the default format for **arecord**. On the other hand, **Audacity** in the same situation - while acknowledging the driver specifications - will also internally convert all captures to the default 'project settings', for which the minimum possible values are 8000 Hz and 16-bit [17].[18]

One of the most important structures is what we could call the 'main' *device* structure, here `minivosc_device`. It can also be a bit difficult to understand, especially since it is - in large part - up to the driver authors themselves to set up the structure, and its relationships to built-in **ALSA** structures. These relationships are of central interest, because a driver author *must* know the location of memory representing the digital audio streams (`snd_pcm_runtime->dma_area` in Fig. B.2), in order to implement *any* digital audio functionality of the driver. And finding this memory location is not trivial - which is maybe best presented in graphical manner, as in Fig. B.2, which shows a partial scope of the 'main' structure `minivosc_device` and its relationships.

On Fig. B.2, only `minivosc_device` has been written as part of the driver code - all other structs (with darker backgrounds) are built-ins, provided by **ALSA**. Pointers are shown on left edge of boxes; self-contained struct variables

[17] which is not the default behavior for actual hardware drivers - they will simply not run some of their predefined callbacks, if the hardware is not present on the bus

[18] While these captures can be exported from **Audacity** as 8-bit, 8 kHz audio files - that process implies an *additional* conversion from the internal 16-bit format.

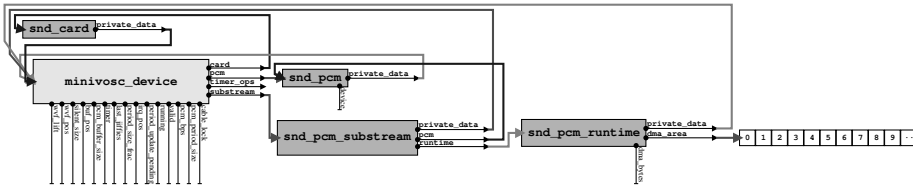


Fig. B.2: Partial 'structure relationship map' of the `minivosc` driver.

are on the bottom edge[19]. Some relationships (such as `snd_pcm_substream->runtime` to `snd_pcm_runtime` pointing) are set up internally by **ALSA**; the relationships to the 'main device' structure (`minivosc_device`) have to be coded by the driver author. Further complication is that the authored relationships can *not* be established at the same spot in the driver code - as some structures become available only in specific **ALSA** callbacks.

This is a conceptual departure from the typical basic understanding of program execution - where a predetermined sequential execution of commands is assumed. Instead, driver programming may conceptually be closer to GUI programming, where the author typically writes *callback functions* that run whenever a user performs some action. Additionally, we can expect to encounter different amount of instances of some of these structs! For example, `snd_pcm_substream` can carry data for a given output connector, which could be stereo. So, if a stereo file is loaded in audio software, and 'play' is clicked - we could expect **ALSA** to pass a *single* `snd_pcm_substream`, carrying data for both channels, to our driver. However, if we are trying to play a 5.1 surround file, which employs 2 stereo and one mono connector - we should expect *three* `snd_pcm_substream`s to be passed to our driver. This could further confuse high-level programmer newcomers, that might expect to receive something like an *array* of substreams in such a case: instead, **ALSA** may call certain callbacks multiple times - and it is up to the driver author to store references to these substreams.

`minivosc` avoids these problems as a mono-only driver - thus within the code, we can expect only one instance of each struct shown on Fig. B.2; and the reference to the only `snd_pcm_substream` can be found directly on the main 'device' struct, `minivosc_device`. This allows us easier focus on another important aspect of **ALSA** - the timing of execution of callbacks, which is necessary for understanding the driver initialization process in general.

[19] Note, the **ALSA** struct boxes show only a small selection of the structs' actual members; while the 'main device' struct still contains some unused variables, leftover from starting example code. Connections are colored for legibility.

Unlike a more detailed UML diagram, a map like Fig. B.2 helps only in a specific context: e.g., the driver is supposed to write to the `dma_area` when the `_timer_function` runs, however this function provides a reference to `minivosc_device`; the map then allows for a quick overview of structure field relationship, so a direct pointer to the `dma_area` can be obtained for use within the function.

B.6 Execution flow and driver functions

The device driver architecture of **Linux** specifies a driver model [16], and within that, certain callback functions that a driver should expose. In the case of **minivosc**, first the **__init** and **__exit** macros ([18, Chapter 2.4]) are implemented, as functions named **alsa_card_minivosc_init** and **alsa_card_minivosc_exit**. These functions run *when a driver module is loaded and unloaded*: the kernel will automatically load modules, built in the kernel, at boot time - while modules built 'out of tree' have to be loaded manually by the user, through the use of the **insmod** program. The **_init** function in **minivosc** registers the driver, and attempts to iterate through the attached soundcards. As **minivosc** is a 'platform' driver, and there is no actual hardware - the **_init**, in this case, is made to always result with detecting a single (virtual) 'card'. Next in line of predefined callbacks are **_probe** and **_remove** [16, 'driver.txt'], in **minivosc** implemented as **minivosc_probe** and **minivosc_remove**. In principle, they would run *when a (soundcard) hardware device is attached to/disconnected from the PC bus*: for instance, **_probe** would run when the user connects a USB soundcard to the PC by inserting the USB connector - if the driver is already loaded in memory. For permanently attached devices (think PCI soundcards), **_probe** would run immediately after **_init** detects the cards; thus, in the case of **minivosc**, **_probe** will run immediately after **_init**, at the moment when the driver is loaded (by using **insmod**).

The **minivosc** driver code informs the system about which are its init/exit functions, by use of **module_init/module_exit** facility (see [19, 'Chapter 2']); while it specifies which are its probe/remove functions through use of the **platform_driver** structure. Finally, last in line of predefined callbacks are the **ALSA** specific callbacks; the driver code tells the system which are these functions, through the predefined **ALSA** struct **snd_pcm_ops**.[20]

While **ALSA** may define more **snd_pcm_ops** callbacks [4], there are 8 of them being used in **minivosc**, by assigning them to functions: one, **snd_pcm_lib_ioctl**, being defined by **ALSA** – and seven **snd_pcm_ops** functions written as part of **minivosc**: **minivosc_pcm_open**, **minivosc_hw_params**, **minivosc_pcm_prepare**, **minivosc_pcm_trigger**, **minivosc_hw_free**, **minivosc_pcm_close**, **minivosc_pcm_pointer**. As clarification - here is the order of execution of above callbacks for the **minivosc** driver, for some common events:

- driver loading: **_init**, then **_probe**
- start of recording: **_open**, then **_hw_params**, then **_prepare**, then **_trigger**
- end of recording: **_trigger**, then **_hw_free**, then **_close**
- driver unloading: **_exit**, then **_remove**

We already mentioned that for the **minivosc** driver, loading/unloading events happen when the **insmod**/**rmmod** commands are executed. 'Start of

[20] Note that the term 'PCM' is used in **ALSA** to refer *generally* to aspects related to digital audio - and *not* to the particular 'Pulse Code Modulation' method as known from electronics (although that is where the term derives from [9, 'PCM (digital audio) interface']).

recording' event would be the moment when the 'record' button has been pressed in **Audacity**; or the moment when we run **arecord** from the command line – correspondingly, 'end of recording' event is when we hit the 'stop' button in **Audacity**; or when **arecord** exits (if, for instance, it has been set to capture for only a certain amount of time). However, note that – *even* with all of this in place – the actual performance of the driver in respect to digital audio is still *not defined*; memory buffer handling is also needed.

B.6.1 Audio data in memory (buffers) and related execution flow

As noted in Sec. B.5 'Driver structures', one of the central issues in **ALSA** driver programming is the location in memory, where audio PCM data for each substream is kept - the **dma_area** field being a pointer to it. In principle, each substream can carry multi-channel data: for instance, a 16-bit sample would be represented as two consecutive bytes in the **dma_area**; while stereo samples could be interleaved [20]. Thus **ALSA** introduces the concept of frames [21], where a *frame* represents the size of one analog sample for all channels carried by a substream. As **minivosc** is specified as a mono 8-bit driver, we can be certain that each byte in its **dma_area** will represent a single sample - and that one frame will correspond to exactly one byte.

The approach to implementing the sampling rate that **minivosc** has (taken from [8]), is to use the **Linux** *system timer* ([22, 'Kernel Mechanisms'], [19, 'Chapter 6']). Note that standard **Linux** system timers are "*only supported at a resolution of 1 jiffy. The length of a jiffy is dependent on the value of HZ in the Linux kernel, and is 1 millisecond on i386* [23]". However, there also exist so-called *high-resolution timers* [24] (for their basic use in **ALSA**, see [7]).

B.6.2 The sound of minivosc - Driver execution modes

The driver writes in the **dma_area** capture buffer repeatedly (as controlled by timers), within the **_xfer_buf** function - or more precisely, within the **minivosc _fill_capture_buf** function called by it. In the **minivosc** code, three different variants can be chosen (at compile time), for copying a small predefined 'waveform grain' array repeatedly in the capture buffer, which results in an audible oscillation when the capture is played back (hence *oscillator* in the name). Note the need to 'wrap' the writing to the capture buffer array, since in **ALSA**, it is defined as a *circular* or *ring buffer* [20]. Finally, all of the three 'audio generation' algorithms can be commented, in which case the **minivosc** driver will simply write a constant value in the buffer. There is an additional facility, called 'buffermarks', which indicate the start and end of the current chunk, as well as the start and end of the **dma_area** - which can be used to visualize buffer sizes.

B.7 Conclusions

The main intent of **minivosc** is to serve as a basic introduction to one of the most difficult issues in soundcard driver programming: handling of digital audio. Given that many newcomers may have previous acquaintance with 'userland' programming, the conceptual differences from user-space to kernel programming (including debugging [1]) can be a major stumbling block. While a focus on capture only, 8-bit / 8 kHz mono driver leaves out many of the issues that are encountered in working with real soundcards, it can also be seen as a basis for discussion of [3a], which demonstrates full-duplex mono @ 8-bit / 44.1 kHz (and can interface with stereo, 16-bit playback). Thus, the main contribution of this paper, driver code and tutorial would be in easing the learning curve of newcomers, interested in **ALSA** soundcard drivers, and digital audio in general.

B.8 Acknowledgments

The authors would like to thank the Medialogy department at Aalborg University in Copenhagen, for the support of this work as a part of a currently ongoing PhD project.

References

[1a] Smilen Dimitrov, "Extending the soundcard for use with generic DC sensors", in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2010)*, Sydney, Australia, Jun. 2010, pp. 303–308, ISSN: 2220-4792, ISBN: 978-0-646-53482-4. URL: http://imi.aau.dk/~sd/phd/index.php?title=ExtendingISASoundcard.

[3a] Smilen Dimitrov and Stefania Serafin, "Audio Arduino - an ALSA (Advanced Linux Sound Architecture) audio driver for FTDI-based Arduinos", in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2011)*, Oslo, Norway, May 2011, pp. 211–216, ISSN: 2220-4792, ISBN: 978-82-991841-7-5. URL: http://imi.aau.dk/~sd/phd/index.php?title=AudioArduino.

[5a] ——, "Towards an open sound card — a bare-bones FPGA board in context of PC-based digital audio", in *Proceedings of Audio Mostly 2011 - 6th Conference on Interaction with Sound*, Coimbra, Portugal, Sep. 2011, pp. 47–54, ISBN: 978-1-4503-1081-9. DOI: 10.1145/2095667.2095674. URL: http://imi.aau.dk/~sd/phd/index.php?title=AudioBareBonesFPGA.

[1] Smilen Dimitrov, "Minivosc homepage", WWW: http://www.alsa-project.org/main/index.php/Minivosc / http://imi.aau.dk/~sd/phd/index.php?title=Minivosc, web page, Last Accessed: 21 December, 2010.

[2] Dave Phillips, "A User's Guide to ALSA | Linux Journal", WWW: http://www.linuxjournal.com/node/8234/print, web page, 2005.

[3] Takashi Iwai, “The ALSA Driver API”, WWW: http://www.alsa-project.org/~tiwai/alsa-driver-api/index.html, web page, Last Accessed: 21 December, 2010.

[4] ——, “Writing an ALSA Driver”, WWW: http://www.alsa-project.org/~tiwai/writing-an-alsa-driver/, web page, Last Accessed: 21 December, 2010.

[5] Ben Collins, “Writing an ALSA driver”, WWW: http://ben-collins.blogspot.com/2010/04/writing-alsa-driver.html, web page, Last Accessed: 21 December, 2010.

[6] Stéphan K., “HowTo Asynchronous Playback - ALSA wiki”, WWW: http://alsa.opensrc.org/index.php/HowTo_Asynchronous_Playback, web page, Last Accessed: 21 December, 2010.

[7] Jaroslav Kysela, “sound/drivers/dummy.c”, WWW: http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.32.y.git;a=blob;f=sound/drivers/dummy.c, web page, Last Accessed: 22 December, 2010.

[8] Jaroslav Kysela, Ahmet İnan, and Takashi Iwai, “drivers/aloop-kernel.c”, WWW: http://git.alsa-project.org/?p=alsa-driver.git;a=blob_plain;f=drivers/aloop-kernel.c;hb=e0570c46e3c4563f38e44a25cfac1f07ff5a02a8, web page, Last Accessed: 22 December, 2010.

[9] Jaroslav Kysela, Abramo Bagnara, Takashi Iwai, and Frank van de Pol, “ALSA project - the C library reference”, WWW: http://www.alsa-project.org/alsa-doc/alsa-lib/index.html, web page, Last Accessed: 22 December, 2010.

[10] www.alsa-project.org, “Asoundrc - AlsaProject”, WWW: http://www.alsa-project.org/main/index.php/Asoundrc, web page, Last Accessed: 22 December, 2010.

[11] Volker Schatz, “A close look at ALSA”, WWW: http://www.volkerschatz.com/noise/alsa.html, web page, Last Accessed: 22 December, 2010.

[12] Matthias Nagorni, “ALSA Programming HOWTO v.0.0.8”, WWW: http://www.suse.de/~mana/alsa090_howto.html, web page, Last Accessed: 15 May, 2011.

[13] Graham Morrison, “Linux audio uncovered”, *Linux Format magazine*, no. 130, pp. 52–55, Apr. 2010, URL: http://www.tuxradar.com/content/how-it-works-linux-audio-explained.

[14] Jaroslav Kysela, “snd-aloop and alsaloop notes”, WWW: http://people.redhat.com/~jkysela/RHEL5/loop/BACKGROUND, web page, Last Accessed: 22 December, 2010.

[15] Jaroslav Kysela and Abramo Bagnara, “git.kernel.org - include/sound/pcm.h”, WWW: http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.32.y.git;a=blob;f=include/sound/pcm.h;, web page, Last Accessed: 22 December, 2010.

[16] git.kernel.org, “Documentation/driver-model”, WWW: http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.32.y.git;a=tree;f=Documentation/driver-model, web page, Last Accessed: 22 December, 2010.

[17] wiki.audacityteam.org, "Bit Depth - Audacity Wiki", WWW: http://wiki.audacityteam.org/wiki/Bit_Depth, web page, Last Accessed: 22 December, 2010.

[18] Peter Jay Salzman, Michael Burian, and Ori Pomerantz, *The Linux Kernel Module Programming Guide*. CreateSpace, 2009, URL: http://linux.die.net/lkmpg, ISBN: 1441418865.

[19] A. Rubini and J. Corbet, *Linux device drivers*. O'Reilly Media, 2001, URL: http://www.xml.com/ldd/chapter/book, ISBN: 0596000081.

[20] Jeff Tranter, "Introduction to Sound Programming with ALSA | Linux Journal", WWW: http://www.linuxjournal.com/article/6735?page=0,1, web page, 2004.

[21] www.alsa-project.org, "FramesPeriods - AlsaProject", WWW: http://www.alsa-project.org/main/index.php/FramesPeriods, web page, Last Accessed: 28 December, 2010.

[22] D.A. Rusling, "The linux kernel", *The Linux Documentation Project*, 1996, URL: http://www.tldp.org/LDP/tlk/.

[23] elinux.org, "High Resolution Timers - eLinux.org", WWW: http://elinux.org/High_Resolution_Timers, web page, Last Accessed: 28 December, 2010.

[24] T. Gleixner and D. Niehaus, "Hrtimers and beyond: Transforming the linux time subsystems", in *Proceedings of the Ottawa Linux Symposium, Ottawa, Ontario, Canada*, URL: http://www.kernel.org/doc/ols/2006/ols2006v1-pages-333-346.pdf, 2006.

Paper C

Audio Arduino - an ALSA (Advanced Linux Sound Architecture) audio driver for FTDI-based Arduinos

Smilen Dimitrov and Stefania Serafin

Refereed article published in the
Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2011), pp. 211–216, Oslo, Norway, May 2011.

The layout has been revised.

Abstract

A contemporary PC user, typically expects a sound card to be a piece of hardware, that: can be manipulated by 'audio' software (most typically exemplified by 'media players'); and allows interfacing of the PC to audio reproduction and/or recording equipment. As such, a 'sound card' can be considered to be a system, that encompasses design decisions on both hardware and software levels - that also demand a certain understanding of the architecture of the target PC operating system.

This project outlines how an Arduino Duemillanove *board (containing a USB interface chip, manufactured by* Future Technology Devices International Ltd *[*FTDI*] company) can be demonstrated to behave as a full-duplex, mono, 8-bit 44.1 kHz soundcard, through an implementation of: a PC audio driver for* **ALSA** *(Advanced Linux Sound Architecture); a matching program for the* Arduino*'s* ATmega *microcontroller - and nothing more than headphones (and a couple of capacitors). The main contribution of this paper is to bring a holistic aspect to the discussion on the topic of implementation of soundcards - also by referring to open-source driver, microcontroller code and test methods; and outline a complete implementation of an open - yet functional - soundcard system.*

Keywords: Sound card, Arduino, audio, driver, ALSA, Linux

C.1 Introduction

A sound card, being a product originally conceived in industry, can be said to have had a development path, where user demands interacted with industry competition, in order to produce the next generation of soundcard devices. As such, the soundcard has evolved to a product, that most of today's consumer PC users have very specific demands from: they expect to control the soundcard using their favorite 'media player' or 'recorder' audio software from the PC; while the soundcard interfaces with audio equipment like speakers or amplifiers. For professional users, the character of 'audio software' and 'audio equipment' may encompass far more specialized and complex systems – however, the expectations of the users in respect to basic interaction with this part of the system is still the same: high-level, PC software control of the audio reproduced or captured on the hardware.

A development of a soundcard thus requires, to some extent, an interdisciplinary approach - requiring knowledge of both electronics and software engineering, along with operating system architecture. But, even with a more intimate understanding of this architecture, a potential designer of a new soundcard may still experience a 'chicken-and-egg' problem: understanding drivers requires understanding of their target hardware - *and* vice versa. As such, considering this product's origins in industry, it is no wonder that literature

discussing implementations of complete 'soundcards' is rare - both hardware and software designs would have to be disclosed, for the discussion to be relevant.

An open soundcard Businesses are, understandably, not likely to disclose hardware designs and driver code publicly; this may explain the difficulty in tracking down prior open devices. It is here that the ARDUINO [1] platform comes into play. Marketed and sold as an open-source product, it is essentially a board which represents a connection between a USB interface chip, and a microcontroller. As the schematics are available, an ARDUINO board can, in principle, be assembled by hand - however, a factory production has both a low, popular price; and brings in a level of expected performance, which allows for easier elimination of problems of electrical nature during development. Thus, on one hand, an ARDUINO board represents *known* hardware - one we could write an **ALSA** driver for; both in principle, and - as this project demonstrates - in reality. On the other hand, the ARDUINO is typically marketed as supporting communication speeds of up to 115200 bps (an impression also stated in [1a]) - which result with data rates, insufficient to demonstrate streaming audio close to the contemporary CD-quality standard (stereo, 16-bit, 44.1 kHz). Yet, the major individual components: FTDI USB interface chip, and ATMEGA microcontroller - are both individually marketed to support up to 2 Mbps: a data rate that can certainly sustain a CD-quality signal. Thus, in spite of being *known* hardware, the ARDUINO may have 'officially unsupported' modes of operation, that would allow it to perform as a soundcard - modes that, however, still need to be quantified in the same sense, as if we were starting to design a board from scratch (*with this particular microcontroller, and USB interface chip*).

Application example An open soundcard may bring actual benefits to electronic instrument designers, beyond the opportunity for technical study: consider a system where a vibrating surface (cymbal) is captured using a sensor and ARDUINO into **PD** software, where it is used to modulate a digital audio signal in realtime. Usual approach would be to read the ARDUINO as a serial port at 115200 bps; this limits the analog bandwidth ($\approx$ 5kHz) and forces the user to code a conversion to **PD**'s audio signal domain; with **AudioArduino** the sensor data could be received directly as a 44.1 kHz audio signal in **PD** - full audio analog bandwidth, no need for signal conversions.

C.2 Previous work

Previous attempts to discuss open soundcard implementations couldn't provide a basis for the development here: the Linux kernel contains many open soundcard drivers, but written for commercial (typically undisclosed) hardware. The now defunct german magazine Elrad may have had a series on implementation

of a PCI card in 1997, but the remaining reference[1] doesn't contain any useful information. The ARDUINO has previously been used for audio: in [2] as a standalone player; [3] as a standalone DSP - but not specifically as a PC-interfaced soundcard. Thus, this project's basis is mostly in own previous work: [1a] demonstrates legacy hardware controlled by PC software; and identifies data throughput control as the main problem in that naïve approach. Modern operating systems address this issue by providing a *driver architecture*; where, in programming a *driver*, the programmer gains a more fine-grained temporal control. In the context of the open **GNU/Linux** operating system(s), acquaintance with its current low-level audio library - **ALSA** - is thus necessary for implementation of soundcard drivers. This project has produced the tutorial driver **minivosc** [2a] as an introductory overview of **ALSA** architecture - also used as a starting point of the work in this paper.

C.3 Degrees of freedom

It would be interesting to qualify to what extent can **AudioArduino** - a system of ARDUINO Duemillanove, microcontroller code, and matching **ALSA** soundcard driver - be considered to be an 'open' 'soundcard system'. To begin with, hardware production necessarily involves mineral extraction and processing, manufacturing, and distribution - stages that require considerable economic infrastructure; and therefore, there will always be a 'hard' price attributed to it. On the other hand software, in essence, represents the instructions - information - for what we can *do* with this hardware. With the increasing affordability causing mass penetration of computing technology, fewer 'hard' investments need to be made to start with software development; and in principle, the pursuit of software development could thereafter involve only investment of the time of the developer. While developer time also carries inherent cost with it, there are circumstances where sharing the outcome - the source code - becomes preferable, for academic, business or altruistic reasons; especially since, with the expansion of the Internet, the physical cost of sharing information can be considered negligible.

Thus, it is in context of software that the term(s) 'free' or 'open' will be applied in this project (as in FLOSS[2]). To begin with, the driver is developed on **Ubuntu** - a FLOSS **GNU/Linux** operating system; with the main corresponding tool for development, **gcc**, being likewise open. The audio framework for **Linux**, **ALSA**, follows the same license - and the main high-level, user audio programs used, **Audacity** and **arecord**, are likewise open. The ARDUINO as a platform is known to be open, by making the schematic files available, as well as offering an integrated development environment (IDE) for **Linux**, which is also open [1]. The microcontrollers used in the platform are typically ATMEGA's, part of the ATMEL AVR family, which (given the tolerance of Atmel to open source, see

[1] http://www.xs4all.nl/~fjkraan/digaud/elrad/pcirec.html
[2] free/libre/open source software

Atmel Application note *AVR911*, also [4]) has long had an open toolchain for programming, **avr-gcc**.

At this point, let's note that Arduino in 2010 released the Arduino UNO board, which is taken to be the 'reference version' for the platform. The reason for this is that the USB interface chip used on the UNO is ATmega8U2, and the USB interface functionality is provided by the open-source **LUFA** (Lightweight USB Framework for AVR) firmware. In contrast, earlier versions of USB Arduinos, like the Duemillanove, feature a FTDI FT232RL USB interface chip. FTDI offers two drivers, VCP (Virtual COM Port, offering a standard serial port emulation) and D2XX (direct access) [5, 'Drivers']. Both of these are provided free of charge - however, source code is not available. Also, VCP may offer data transfer rates up to 300 kilobyte/second, while D2XX up to 1 Megabyte/second ([5, 'Products/ICs/FT245R']). Nonetheless, there exists a third-party open-source driver for FTDI in the **Linux** kernel, which corresponds to VCP, named **ftdi-sio** [6] - in fact, **ftdi-sio** forms the basis of the **AudioArduino** driver. With this, the following parts of the **AudioArduino** system can be considered open: *microcontroller code*, and tools to implement/debug it; *audio driver*, and tools to implement/debug it; *operating system*, hosting the development tools, the driver and high-level software; and *high-level audio software*, needed to demonstrate actual functionality – i.e., the bulk of the software domain. The driver was developed on **Ubuntu** 10.04 (Lucid), utilizing the 2.6.32 version of the **Linux** kernel; the code has been released as open source, and it can be found by referring to the home page [7].

C.4 Concept of AudioArduino

Given that the ATmega328 features both ADC, and DAC (in form of PWM), converters - using the Arduino as a soundcard hardware is a feasible idea, as long as one trusts that the data transfer between the PC and the ATmega328 can occur without errors at audio rates. Developing a USB driver for such data transfer would, essentially, require a good working knowledge of the USB bus and its specifications. However, that is a daunting task for any developer - the USB 2.0 Specification [8] alone is 650 pages long; with actual implementation, in a form of a driver for a given OS, requiring additional effort. Therefore, the starting point of this project is to abstract the USB transport to the greatest extent possible, and avoid dealing with particular details of the USB protocol. This is possible because of the particular architecture of the Arduino board, rendered on Fig. C.1.

As Fig. C.1 shows, the **ftdi-sio** driver makes the FT232 device appear as a 'serial port' in the PC OS, that the user can write arbitrary data to. The driver will format this data as necessary for USB transport, and send it on wire; the FT232 will then accept this data and convert it to TTL-level (0-5V) RS-232 signal (and the same happens for the reverse direction, when reading). Given that RS-232 is conceptually much easier to understand (e.g., [9]); we can

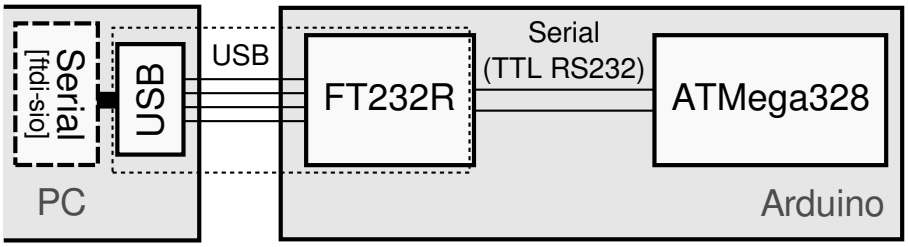


Fig. C.1: Simplified context of an ARDUINO, connected to a PC.

'black box' (abstract) the *unknown* (USB) part in the data transfer - and focus on the *known* (RS-232) part.

In order to specify what sampling rates, in terms of digital audio, would this hardware support - the most important factor to consider is the data transfer rate, that can be achieved between the ATMEGA328 and the FT232 over the serial link. As far as this serial link goes, the ATMEGA328 states maximum rate of 2.5 Mbps [10, pg.199]; while the FT232 states up to 3 Mbaud [11, pg.16]. As the **ftdi-sio** driver supports 2 Mbps[3] by default, this is the 'theoretical' speed that should be possible to achieve all the way through to the ATMEGA328. A speed of 2 Mbaud translates to 200000 Bps[3], which would be enough to carry 200000/44100 = 4.5 mono/8-bit/44.1 kHz channels; or two mono/16-bit/44.1 kHz channels; or one CD quality stereo/16-bit/44.1 kHz channel. However, one still needs to determine what *actual* data transfer rates can be achieved, and under which conditions (such as different software). Beyond this, it is the response times of the ATMEGA328 (including DAC and ADC elements), that would limit the use as full-duplex device. The final issue is the analog I/O interface, discussed further in this paper.

Building and running Both the source code, and instructions for building and running, can be found in [7] (and they are similar to those given in [2a]). The source code consists of a modified version of [6], **ftdi_sio-audard.c**; the **ALSA**-specific part in **snd_ftdi_audard.h**; associated headers and a **Makefile**; and microcontroller code, **duplexAudard_an8m.pde**. The **.pde** code can be built and uploaded to the ARDUINO using the **Arduino IDE**.

With this in place, high-level audio software (like **Audacity**) will be able to address the ARDUINO, and play back and capture audio data through it. ARDUINO's analog input 0 (**AIN0**) is treated as a soundcard input; sensors (like potentiometers) connected to this input can have their signal captured at 44.1 kHz in audio software. ARDUINO's digital pin 6 (**D6**) is soundcard output; on

[3]Note that in 8-N-1 RS232 transfer, there are 8 data bits, 1 start and 1 stop bit; so 8-bit data is carried by 10-bit packet. Usually, 'baud' means 'signal transitions per second' and refers to all 10 bits, while 'bps' as 'bits per second' should refer to the 8 data bits only; but they can be often used interchangeably - 'Bps' as 'bytes per second' refers strictly to data payload (see also [12]).

which, when audio software plays back audio data, (analog) PWM output is generated (audible).

C.5 Quantifying throughput rate - duplex loopback

As mentioned, one of the biggest issues in estimating if the ARDUINO board can behave as a soundcard, is in measuring the actual data transfer rate that can be achieved. The initial question is what tools can be used for that: the **ftdi-sio** driver will make a connected ARDUINO appear as a special file in the **Linux** system (`/dev/ttyUSB0`), representing a serial port. The serial port settings, such as speed, can be changed by using the **stty** program. Thereafter writing character data to the ARDUINO can be performed by writing to the associated file, say, by using `echo 'some text' > /dev/ttyUSB0` - and reading by, say, `cat /dev/ttyUSB0`.

However, finding the actual data rate in either direction is not the only thing which is interesting; another interesting point is to what extent can the ARDUINO board be considered a *full-duplex* device; i.e., whether the device can both receive and send data *simultaneously* (which, in terms of soundcards, is a standard expected behaviour). To assess both points, we suggest the ATMEGA328 is programmed as a 'digital loopback': to listen for incoming serial data; and send back the received byte through serial, as soon as it has been received. Then for the PC side, we propose a simple threaded program, **writeread.c** [12]: it accepts an input file; initiates write and read operations on a serial port in separate threads, so they can run concurrently; writes the input file, and saves the received data in another; and times these operations, so that the throughput rate can be determined.

What this experiment shows, is that the usual `C` commands for reading and writing from a serial port (and by extension, user programs like `cat` or `echo`) do not carry the concept of a data rate - they simply try to transfer data as fast as possible; and even for 2 Mbps communication, these commands push data faster than the USB chip can handle, which results with kernel warnings. Therefore, it is up to the program author to implement some sort of buffering, that would provide an effective throughput rate. Yet even with this in place, limiting rate to 2 Mbps within **writeread.c** would *still* cause throttling warnings; but, limiting it to slightly *below* 2 Mbps allows for a error-less demonstration. The reason for this is likely in the asynchronous nature of the serial RS232 protocol: in not sharing a single clock; the PC, the FT232 and the ATMEGA328 each have a slightly different concept of what the basic time unit (clock tick) duration would be - and thus a different concept of what '2 Mbps' is. By lowering the data rate from **writeread.c**, we likely account for these differences, which allows for error-free transmission; and from the PC, we can typically measure around 98% of 2 Mbps achieved for error-free duplex transmission.

Moreover, during this digital loopback experiment, the signals of the `TX` and

RX connections (between the FT232 and the ATmega328) were measured with an Agilent 54621A[4] oscilloscope; captured with the open-source **agiload** for **Linux**; and analysed using a script produced by this project, written in **python** (utilizing **matplotlib**) that features a serial decoder, called **mwfview-ser.py** [7]. These measurements show that the time for the ATmega328 to receive a byte and send it back - the minimal 'quantum' of action, relevant for a 'digital duplex' - is around 6.940 μs (Fig. C.2), which is approx. 31% of the 22.6 μs analog sample period (for 44.1 kHz rate); which specifies the latency bottleneck expected from the Arduino in 'digital loopback' mode.

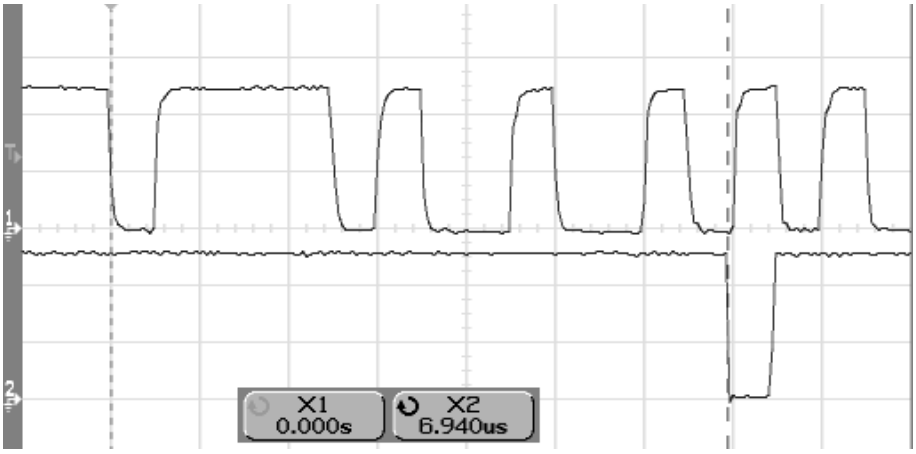


Fig. C.2: Oscilloscope capture of RX (top) and TX (bottom) serial lines at the ATMega328, indicating latency between received and sent byte.

Note that, the ATmega328's UART produces a signal with considerably more jitter than the FT232[5]; and there can be gaps in the otherwise sustained rate of serial transmission between the two - but none of this seems to harm error-free transmission at 2 Mbps. Finally, **writeread.c** works both with the 'vanilla' **ftdi-sio** driver, and the **AudioArduino** driver. Also, the same Arduino code used to demonstrate digital loobpack with **writeread.c**, can be used with the **AudioArduino** driver - allowing for demonstration of a *digital audio loopback*: one can load a file in **Audacity**; play it back through the **AudioArduino** card; and by recording at the same time from the same card, one should capture the very same audio being played back (latency notwithstanding).

[4]The Agilent 54621A claims 60 MHz bandwidth, which is sufficient for capture of a 2 Mbps digital signal

[5]A crude measurement of jitter spans around 0.26 μs, which is about 52% of the 0.5 μs period for a bit transition at 2 Mbps, see [12]

C.6 Microcontroller code

There are two distinct versions of microcontroller code for the ATmega328 used in this project, both in a form of a C language .pde file (the default format compilable in the Arduino IDE). The first is the mentioned 'digital duplex' code, which simply sends back any byte received through serial, posted in [12]. The main issues here are: the setup of the ATmega328's UART to support 2 Mbps (which is not supported in the default Arduino API); removing all overhead due to API function calls, by using the function source code directly; and disabling all irrelevant interrupts - before the Arduino can start showing 98% of 2 Mbps with **writeread.c**. Beyond this, the code can be implemented either as a single loop, or with interrupts on incoming serial data; with no significant difference in respect to performance. This is the same microcontroller code used as basis for development of the **AudioArduino** driver.

Once the **AudioArduino** driver was confirmed to be working with the 'digital duplex' code - a new, second 'analog I/O' version was written, which also employs the ADC and PWM (as DAC) facilities of the ATmega328. This version, as it is supposed to support audio playback and recording, requires deeper involvement with the ATmega328 datasheet [10]. In essence, the problem is that **ALSA** will send (mono) data at rate of 44100 Bps, which will appear as chunks of bytes on the 200000 Bps serial Arduino line; these bytes need to be stored as soon as possible by the ATmega328 in memory (buffer). On the other hand, at a rate of 44100 Hz, the ATmega328 should read one byte from the buffer and write it to PWM (the DAC) - and at the same time, read a byte from the ADC, and send it via serial. As we would expect an 8-bit interface (where each byte represents an analog sample) at the driver side, no further digital sample processing needs to be done in either direction. This is solved by code that employs an interrupt on incoming data, where the data is stored in a circular buffer - and a (16-bit) timer interrupt to handle the analog I/O at the 44100 Hz analog rate [7]. Note that this 'analog I/O' version seems to only perform well when implemented with incoming data handled on interrupt; trying to do the same handling in a single loop reveals problems with determining *when* an incoming byte is ready to be read from ATmega's UART [7].

C.7 Driver architecture

The **AudioArduino** driver is not only based on **ftdi-sio** - **ftdi_sio-audard.c** is a renamed version of [6], with several changes: first, it includes **snd_ftdi_audard.h**, which here is not used in the standard sense of a C header, but simply as a container for **ALSA** relevant code (which would, otherwise, have to be written into the already complex [6]). Other changes include calling **ALSA** relevant functions from the default **ftdi-sio** functions: `audard_probe` from `ftdi_sio_probe`; `audard_probe_fpriv` from `ftdi_sio_port_probe`; `audard_remove` from `ftdi_sio_port_remove`; and `audard_xfer_buf` from `ftdi_process_`

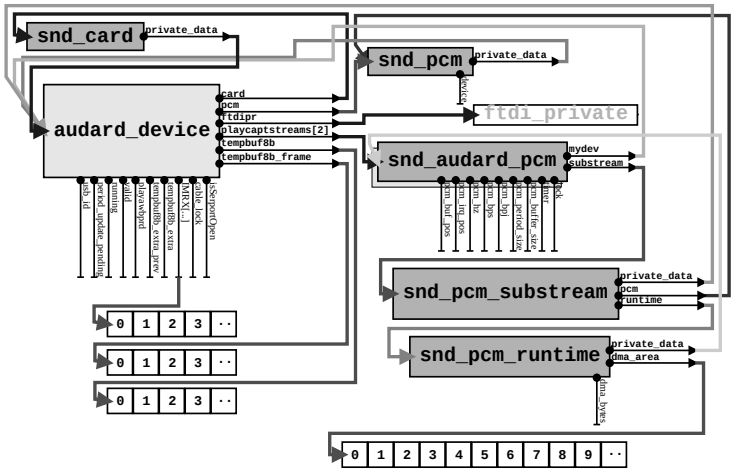


Fig. C.3: Partial 'structure relationship map' of the `AudioArduino` driver.

`packet` - which connects the soundcard **ALSA** interface to USB events.

Otherwise, the main **ALSA** functionality is contained in `snd_ftdi_audard.h`, whose development is based on `minivosc.c` [2a]. Thus, it contains the same type of **ALSA** related structures, but the structure map (shown on Fig. C.3) is slightly more complex than in [2a]: the main 'device struct', `audard_device`, contains an array holding references to both the playback and the capture substream; the substreams are encapsulated in `snd_audard_pcm` structures, that hold individual buffer position counters. There are separate `snd_pcm_hardware` and `snd_pcm_ops` variables - yet a single `snd_card_audard_pcm_timer_function` - to handle the playback and capture substreams.

In essence, the `AudioArduino` driver leaves, for the most part, the functionality of `ftdi-sio` as is; with several additions. When `ftdi_sio_probe` runs (i.e., when the Arduino is connected to PC via USB), the **ALSA** interface is additionally setup, enumerating the Arduino as a soundcard. With this in place, on one hand, the driver keeps the serial interface (such as the creation of the `/dev/ttyUSB0`) file. On the other hand, the driver will also react on 'start' or 'stop' commands from high-level audio software as usual: e.g., on 'start' `_trigger` will run, which will start the timer, and thus the periodic calls to `_timer_function`. The `_timer_function`, then, needs to handle the playback direction by copying the respective part of its `dma_area` to USB - which it does by calling `ftdi_write`. For the capture direction, incoming USB data triggers `ftdi_process_packet`, which additionally calls `audard_xfer_buf`; here USB data is copied to a dynamically sized 'intermediate' buffer, `audard_device->IMRX` – and `_timer_function` will thereafter copy the data from the intermediate buffer to the capture substream's `dma_area`, the next time it runs.

The `AudioArduino` driver additionally exposes CD quality, stereo/16-bit/44.1kHz capability - to allow for direct playback interface with `Audacity` (and most media player software). However, since the microcontroller code expects

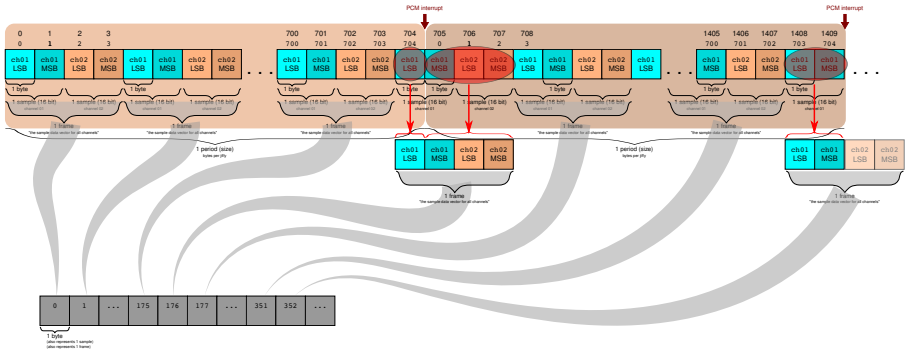


Fig. C.4: Visualisation of driver's playback buffer boundaries, and CD to mono/8-bit conversion.

a sequence of 8-bit values, we must convert the stereo 16-bit stream to a mono 8-bit one - this opens a whole new set of problems related to wrapping, which is illustrated on Fig. C.4. By declaring the driver capable of 16-bit stereo, we have not changed the number of substreams (which would correspond to connectors on the soundcard); however, Fig. C.4 shows that we would have changed the data format carried in the substream's `dma_area` - the stream is now interleaved: consecutive bytes carry a pattern of left channel's 2 bytes, followed by right channel's 2 bytes. Thus an **`ALSA`** *frame* (size of analog sample in all channels) is now 4 bytes; and the problem becomes how to represent this **`ALSA`** frame with a single byte. The approach in the **`AudioArduino`** driver is to simply extract the most significant byte of the left channel, according to the formula (`C` code): `(char) (left16bitsample >> 8 & 0b11111111) ^ 0b10000000`

However, as Fig. C.4 shows, a bigger problem is that the wrapping boundaries (at the size of the chunk handled at each `_timer_function`, and at the size of `dma_area`) can now occur in the *middle* of a frame (and correspondingly, middle of an 8-bit sample) - which is a situation that doesn't occur for 8-bit streams (where each single byte corresponds to one analog sample). To address this, the **`AudioArduino`** driver employs yet another intermediate buffer (`audard_device->tempbuf8b`). With this in place, the driver will automatically convert a 16-bit stereo stream from **`Audacity`** to an 8-bit one, preserving the 44100 Bps rate, before it sends it to USB - and thus, an audio 'digital loopback' can be demonstrated on this driver directly from **`Audacity`**.

Finally, note that 'DMA' in '`dma_area`' stands for 'Direct Memory Access', which "*allows devices, with the help of the Northbridge, to store and receive data in RAM directly without the intervention of the CPU (and its inherent performance cost)* [13]". Interestingly, in this case: while the transfer of incoming USB data to PC memory (as part of **`ftdi-sio`**); as well as the transfer of data from `dma_area` to user memory of high-level audio software (as part of the **`ALSA`** 'middle layer'); likely involves DMA – the transfer of memory that is performed as part of **`AudioArduino`**'s `_timer_function` definitely *doesn't*; as we use the `memcpy` command to transfer data (which does involve the CPU).

C.8 Analog I/O

The **ALSA** driver can be developed in its entirety with the 'digital duplex' Arduino code; if thereafter the 'analog I/O' microcontroller code is 'burned' on the Arduino - the driver will, effectively, utilize analog input pin 0 as analog input connector, and digital pin 6 as analog output connector. However, both the analog input range, and the output PWM signal, span the voltage range from 0 to 5V - while a typical off-the shelf soundcard typically contains 'line' input and output connectors, as well as 'mic in' and 'speaker out' connectors, which follow a different analog standard. These topics are discussed in more detail in an associated paper, [4a].

The use of analog pins on the Arduino to read sensors is standard practice, and plenty of examples can be found on the web [1]; thus an arbitrary sensor signal can be captured through high-level audio software at 8-bit, 44.1kHz quality (in the same spirit of [1a]). Note that the analog input voltage range, 0-5V, will be represented with the span of 8-bit values from 0 to 255 - which within **Audacity** may be treated as floating point values -1 and 1, respectively.

The use of PWM to deliver an analog audio signal is based on the premise that the highest PWM frequency obtainable from the Arduino, 62500 Hz [4a], will be sufficient to reproduce a 44100 Hz digital (22.05 kHz analog) audio signal. To a novice, used to analog voltage waveforms, this can be problematic to assess - as the binary nature of PWM makes it seem inherently 'distorted' in the time domain. However, industry insiders are well aware of the practice of using PWM for audio, e.g., in the mobile or automotive industry [14], and often to drive speakers directly [15]. This project demonstrates that as well: upon playback of audio from high-level software, one can simply connect the output pin 6 to a channel on headphone jack, and connect the ground of the headphone jack to Arduino's ground - and audible sound would be perceived from the headphones' speaker (but use of a capacitor will result with a louder, clearer sound [7]). Note that there are inherent jitter problems in reproducing HF tones with this technique, while mid-range music can be reproduced with acceptable quality [4a], [7].

C.9 Conclusions

As this paper outlines, development of a soundcard can be a complex and involved issue. The particular approach used here, avoids many electronic engineering issues by choosing the Arduino Duemillanove as soundcard hardware; and avoids deeper involvement with the USB protocol by the specific use of the **ftdi-sio** driver as a basis. In doing that, the overview of the **ALSA** architecture, started in [2a], is finalized - as **ALSA** is discussed in its full intended scope: in relation to a given soundcard hardware, and given interface bus. This allows for focus on issues in soundcard implementation that are close to 'first principles', and as such could serve in educational context, as a basic introduction

to newcomers to the field - which is the main contribution of this paper and source code.

Beyond (hopefully) furthering the discussion on DIY implementations of PC interfaced digital audio hardware, this project may have a practical impact as well - as there are research projects in the computer audio community and related fields (such as haptics [16]), which use the ARDUINO to capture sensor data; and as such, could benefit from the audio-rate capture quality, and the possibility to leverage the real-time performance of applicable high-level audio software, such as **PureData**.

C.10 Future work

The current **AudioArduino** code could, in principle, easily be modified to demonstrate stereo 8-bit performance, or even 16-bit mono (say, by using separate PWM for LSB and MSB, and mixing them in the analog domain). A more involved work would be to port the concept to the reference ARDUINO UNO - as that will require work on the **LUFA** firmware, which doesn't currently support 2 Mbps[12]; on the other hand, the **LUFA** could allow the ARDUINO to be recognized as a 'USB audio' class device, instead of a 'USB serial' one. Finally, as in [2a], it would be interesting to see to what degree could **AudioArduino** be ported to the major proprietary PC operating systems.

C.11 Acknowledgments

The authors would like to thank the Medialogy department at Aalborg University in Copenhagen, for the support of this work as a part of a currently ongoing PhD project.

References

[1a] Smilen Dimitrov, "Extending the soundcard for use with generic DC sensors", in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2010)*, Sydney, Australia, Jun. 2010, pp. 303–308, ISSN: 2220-4792, ISBN: 978-0-646-53482-4. URL: http://imi.aau.dk/~sd/phd/index.php?title=ExtendingISASoundcard.

[2a] Smilen Dimitrov and Stefania Serafin, "Minivosc - a minimal virtual oscillator driver for ALSA (Advanced Linux Sound Architecture)", in *Proceedings of the Linux Audio Conference (LAC 2012)*, Stanford, California, USA, Apr. 2012, pp. 175–182, ISBN: 978-1-105-62546-6. URL: http://imi.aau.dk/~sd/phd/index.php?title=Minivosc.

[4a] ——, "An analog I/O interface board for Audio Arduino open soundcard system", in *Proceedings of the 8th Sound and Music Computing Conference (SMC 2011)*, Padova, Italy: Padova University Press, Jul. 2011, pp. 290–297, ISBN: 978-8-897-38503-5. URL: http://imi.aau.dk/~sd/phd/index.php?title=AudioArduino-AnalogBoard.

[1] arduino.cc, "Arduino homepage", http://www.arduino.cc/, web page.

[2] Limor Fried, "ladyada.net Wave Shield - Audio Shield for Arduino", WWW: http://www.ladyada.net/make/waveshield/, web page, Last Accessed: 29 December, 2010.

[3] Martin Nawrath, "Arduino Realtime Audio Processing", WWW: http://interface.khm.de/index.php/lab/experiments/arduino-realtime-audio-processing/, web page, Last Accessed: 29 December, 2010.

[4] S. Wilson, M. Gurevich, B. Verplank, and P. Stang, "Microcontrollers in music HCI instruction: reflections on our switch to the Atmel AVR platform", in *Proceedings of the 2003 conference on New interfaces for musical expression*, Citeseer, 2003, pp. 24–29.

[5] www.ftdichip.com, "FTDI Homepage", WWW: http://www.ftdichip.com/, web page, Last Accessed: 29 December, 2010.

[6] Greg Kroah-Hartman, Bill Ryder, and Kuba Ober, "drivers/usb/serial/ftdi_sio.c", WWW: http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.32.y.git;a=blob;f=drivers/usb/serial/ftdi_sio.c, web page, Last Accessed: 29 December, 2010.

[7] Smilen Dimitrov, "AudioArduino homepage", web page, Last Accessed: 21 December, 2010. URL: http://imi.aau.dk/~sd/phd/index.php?title=AudioArduino.

[8] www.usb.org, "USB.org - Documents [Specifications home]", WWW: http://www.usb.org/developers/docs/, web page, Last Accessed: 29 December, 2010.

[9] Smilen Dimitrov and Stefania Serafin, "A simple practical approach to a wireless data acquisition board", in *Proceedings of the 2006 conference on New interfaces for musical expression*, IRCAM-Centre Pompidou, 2006, pp. 184–187, ISBN: 2844263143.

[10] www.atmel.com, "Atmel ATmega48A/48PA/88A/88PA/168A/168PA/328/328P datasheet", WWW: http://www.atmel.com/dyn/resources/prod_documents/doc8271.pdf, web page, Last Accessed: 29 December, 2010.

[11] www.ftdichip.com, "FT232R USB UART IC Datasheet Version 2.07", WWW: http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf, web page, Last Accessed: 29 December, 2010.

[12] www.arduino.cc, "Arduino Forum - Measuring Arduino's FT232 throughput rate ?", WWW: http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1281611592/0, web page, Last Accessed: 29 December, 2010.

[13] Ulrich Drepper, "What every programmer should know about memory", Nov. 21, 2007. URL: http://people.redhat.com/drepper/cpumemory.pdf (visited on 05/02/2014).

[14] Mikkel Christian Wendelboe Høyerby, Michael Andreas E. Andersen, Dennis R. Andersen, and Lars Petersen, "High Bandwidth Automotive Power Supply for Low-cost PWM Audio Amplifiers", English, in *NORPIE2004*, Trondheim, 2004. URL: http://server.oersted.dtu.dk/publications/views/publication_details.php?id=1000.

[15] Finn T. Agerkvist and Lars M. Fenger, "Subjective test of class D amplifiers without output filter", English, in *117th Audio Engineering Society Convention*, 2004.

[16] Luca Turchet, Rolf Nordahl, Stefania Serafin, Amir Berrezag, Smilen Dimitrov, and Vincent Hayward, "Audio-haptic physically-based simulation of walking on different grounds", in *Proceedings IEEE Multimedia Signal Processing Conference (MMSP'10)*, Stéphane Pateux, Ed. Saint Malo, France: IEEE Press, 2010, pp. 269–273, ISBN: 978-1-4244-8110-1. DOI: 10.1109/MMSP.2010.5662031.

Paper G

Comparing the CD-quality, full-duplex timing behavior of a virtual (dummy), hda-intel, and FTDI-based AudioArduino soundcard drivers for Advanced Linux Sound Architecture

Smilen Dimitrov and Stefania Serafin

Manuscript submitted to / in review for
Linux Journal, 2015.

The layout has been revised.

Abstract

Our previous work within soundcards in an open source context ([1a, 2a, 3a, 4a, 5a]) deals primarily with low fidelity audio reproduction (up to 8 bit, 44.1 kHz, mono). The next natural step would be to address high fidelity reproduction, where we would primarily focus on Compact Disc quality (16 bit, 44.1 kHz, stereo). However, looking on the PC operating system side, the transition from our previous lo-fi drivers to CD-quality ones is challenging, as it requires an understanding of the preemptive nature of the OS kernel, whose influence makes especially full-duplex operation challenging. This lead us to a comparison, discussed in this paper, between the full-duplex, CD-quality operation of ALSA drivers: a modified virtual ("dummy") driver, and a HDA Intel onboard soundcard driver; for the 2.6.3 series of Linux kernels (same used as in our previous work). In light of this, we also discuss the possibility of developing a full-duplex, CD-quality version of our AudioArduino ALSA driver (intended for the older FTDI-based Arduinos); as well as the software tools and debug analysis approaches used during development (some of them developed by ourselves, and released as open-source [1]).*

Keywords: audio systems, open systems, Linux kernel, computer science, driver, ALSA

G.1 Introduction

In our previous work, we have explored the personal computer's (PC) operating system (OS) perspective on the soundcard as a distinct peripheral device, by focusing on implementations of soundcard drivers for the Advanced Linux Sound Architecture (ALSA): a virtual, capture-only, 8 bit/8 kHz/mono driver called `minivosc` in [2a]; and a Universal Serial Bus (USB), full-duplex, 8 bit/44.1 kHz/mono driver called AudioArduino in [3a] (also reused in [5a]). Since one of the main motivations behind this work, is the exploration of the potential of open source soundcard hardware and software, as a basis for development of undergraduate laboratory exercises in real-time streaming, on the scale of a typical do-it-yourself (DIY) enthusiast [6a] – our approach was to identify and discuss crucial points from the high-level software and hardware development process, while attempting to abstract the low-level details (such as the physical layer of USB communication, or the exact operation of the OS kernel) as much as possible.

In retrospect, this choice of low fidelity audio transfer data rates (8000 and 44100 B/s, respectively) was the key aspect allowing us to focus on basics such as the general layout of an ALSA driver, or the mechanics of the driver's circular buffer wrapping, while assuming that the rest of the system will simply execute the timing right - and in that, having the freedom to ignore the details of what goes on behind the scenes in the OS. On the other hand, we were also

motivated to extend the exercise into what is known as the Red Book Compact Disc Digital Audio (CD-DA) format [2]: 16 bit/44.1 kHz/stereo (or a data rate of 176400 B/s), not only because it has been considered a standard for high-quality audio in the past decades, but also because it introduces the concept of multichannel audio (even if that multichannel operation is represented by only two, the left and right, channels) - which we find to be a potentially exciting laboratory exercise.

Our constraints were: to remain within the same kernel series as the rest of our soundcard projects, which were developed on Ubuntu 10.04 Lucid (based on Linux kernel version 2.6.32) and Ubuntu Natty (kernel 2.6.38) GNU/Linux operating systems; to use relatively modern, but low-end development computers (which in our case reduces to two netbooks, MSI Wind U135 and a NetColors brand, both with dual Intel Atom N450 1.67 GHz processors; at least 1 GB RAM; and hard disk drives, as opposed to SSD); and to eventually come up with a driver product, that can demonstrate CD-quality full-duplex operation with standard audio user software like Audacity. However, we quickly found that within these constraints, this transition of a driver to higher data rates is far from trivial, if one is not thoroughly acquainted with the potential obstacles. To begin with, both of our previous drivers are derived from the `dummy` (or `snd-dummy`) virtual driver from ALSA, which uses the standard Linux timer functions to handle capture and playback operations. These Linux timer operations, however, have a resolution of a *jiffy*, which, depending on the platform, can be up to several milliseconds in duration; and the OS kernel can decide to reschedule a timer function at any time, by which audio drops can be introduced in the audio streams. With a move to the high-resolution timer application programming interface (API) in Linux, we arrived at a virtual driver that exhibited no problems with ALSA-only programs such as `aplay` or `arecord` - but, when used with Audacity in full-duplex mode, it triggered a very specific error in the underlying PortAudio user-space library. This lead us to perform a comparison between the virtual driver, and a driver for an on-board HDA Intel soundcard - and in that, recognize that ALSA, as a programming interface, is primarily oriented towards card hardware that can utilize DMA (direct memory access), and otherwise communicate to the PC host through hardware interrupts. Note that the original `dummy` driver (unlike our `minivosc` and AudioArduino ones) does not perform any memory copying operations, and thus needs to spend no time on them; therefore the study of this driver alone doesn't suffice for analyzing streaming errors, as `dummy` cannot be made to exhibit them (like the ones described later in this article).

Software timers (even high-resolution ones) add a layer of unreliability and delays, and as such cannot ideally simulate the operation of a DMA interrupt card. However, by attempting to simulate the operation of the HDA Intel driver in our virtual driver with timer functions, we still managed to arrive at a virtual driver called `dummy-fix` [1], that performs memory operations (writing in the capture stream, like `minivosc`) and can demonstrate a full-duplex CD-quality operation in Audacity, without problems and on the order of minutes.

Unfortunately, this alone is not readily transferable to our AudioArduino driver for FTDI-based Arduino boards (like the Arduino Duemilanove): the kernel functions for USB transfer, can have a timing as unreliable as software timers. This, on one hand, can cause detection of missing bytes as over- or underruns in e.g. the PortAudio user-space library (and subsequent stream restarts) - and on the other hand, can cause an overrun error in the internal buffers of the FTDI FT232 USB-serial chip, which is even harder to debug, due to the closed and proprietary nature of the chip's specifications.

In the scope of this article, we will only denote the base of decimal numbers if needed to distinguish them from other notations; binary numbers (base 2) will be indicated by a subscript suffix; while for hexadecimal numbers (base 16), instead of a subscript suffix, we will use the prefix "0x" (common in programming languages like `C`); e.g. $107_{10} = 01101011_2 = 0x6B$. We will use a `teletype font` to indicate software related concepts like variables, kernel- and user-space functions (which may occasionally be suffixed by a pair of parentheses, to emphasize their role as a function), files, libraries, packages, scripts and programs – though typically not languages or applications commonly known by their capitalized name. We will also use "OSs" as acronym for the plural form of "Operating Systems"; and to keep it disambiguated from "Open Source Software", we will refer to the that concept as free/libre/open-source software (FLOSS) instead; conversely, we will denote non-free, gratis software as "freeware".

This article essentially details the development chronology outlined thus far, and is organized as follows: the introduction continues with section G.2, which provides a basic introduction to the issue of kernel preemption; followed by section G.3, which discusses the behavior of timer functions in the Linux kernel. This leads into section G.4, which argues the reasons why our previous work didn't show any problems, even in presence of timing function jitter. During this phase, we developed a small Python-based application to facilitate browsing of timestamped log files, discussed in subsection G.4.1. Section G.5, on the whole, describes the development of a virtual, CD-quality, full-duplex driver for ALSA; in more detail, subsection G.5.1 provides an overview of the different components interacting through ALSA, emphasizing our development context. The following subsection, G.5.2, reviews the concept of full-duplex, and related ALSA concepts; while the next subsection G.5.3 first focuses on the concept of DMA, and then discusses the procedure we used to compare the full-duplex operation of the HDA Intel and dummy ALSA drivers. This is followed by subsection G.5.4, where we note our use of the open source plotting application `gnuplot` to visualize Linux kernel logs, and how the plot analysis led to a solution for a virtual CD-quality full-duplex driver. The difficulty in applying the fixes from our full-duplex virtual driver, to a driver for an FTDI-based Arduino, is expounded on in section G.6. First, we take a closer look at the mechanics of USB, and the concept of full-duplex therein, in subsection G.6.1. Then, in subsection G.6.2, we provide an overview of our analysis, that lead us to conclude that the source of the particular error is an overrun in the buffers

on board the FT232 chip. Finally, we detail the inconclusive analysis efforts to solve this error in subsection G.6.3, which also includes visualization aspects: a special difficulty during the profiling of the FT232 chip, was the need to plot multiple tracks of relatively fine-grained data with a relatively long duration - we addressed this with a development of a small Python-based application, also discussed in this subsection. We then provide an overview of some of the debugging approaches we used in section G.7, followed by a general comment on obsolescence in section G.8; and we finally wrap up with a conclusion in section G.9. Most of the driver code and scripts developed by us, used for discussions in this article, have been released as free/libre/open-source software; please see [1] for more.

This work was originally intended for the computer and electronic music instrument community; the multidisciplinary research in this area usually abstracts the technical operation of the soundcard without focusing on it. Thus, arriving at a level allowing for design within digital audio hardware, would require consultation of literature that assumes acquaintance with concepts, not central to this background. For this reason, we have tried to focus (as in sect. G.2), on concepts that might be considered trivial from the perspective of a computer or signal processing scientist; or detail debugging and analysis approaches and tools (as in sect. G.7), that might be trivial to an experienced engineer developer. We believe that one of the main issues in streaming digital audio design is the ability to predict the possibility of errors (like overruns) from the performance characteristics of participating devices, already in the design stage. With the tutorial-like approach, which includes details on errors we encountered in the development process, we hope to contribute with a basic reference - with reproducible, FLOSS, examples - that will support better understanding of these issues, in a multidisciplinary context outside of their native fields.

G.2 A basic understanding of Linux kernel operation and preemption

On the simplest conceptual level, a software program can be understood as a sequential, ordered list of software commands, as afforded by a particular programming language. This implies that the commands the programmer writes, are going to be executed in the same sequential order by the processor of a computer. This general idea holds for both low-level (e.g. `C`) and high-level (e.g. scripting, like JavaScript) languages, even if it is immediately rendered inaccurate, at the least, by the very existence of facilities for unconditional (e.g. `jump`) and conditional (e.g. `if/else`) branching in most programming languages. However, we can also consider that programs typically branch to a group of sequential commands, which can be rudimentarily equated to the concept of a subroutine (a.k.a. function); and for a sequence of non-branching commands within any subroutine, we usually take for granted that they will

be executed in that same order.

However, this simple conceptualization does not reveal exactly *what* is executed by the computer processor *when* – that is, it doesn't supply us with any model for the timing of execution. The "what" part of the question is somewhat easier to answer, since what a central processing unit (or CPU) executes is a machine instruction: in essence, a sequence of integer numbers. While these numbers are often written in hexadecimal notation, their bit pattern in binary notation would, essentially, represent the voltage pattern enforced on the wires (or lines) of the address and data busses connected to the CPU pins at a particular moment in time. As such, machine instructions are particular to each CPU architecture, and its engine as implemented *in silico* - and the changes of the voltage pattern they represent, in *takt* with the oscillator clock that drives the CPU, would represent (a part of) the execution of software in the physical world.

The difficulties of programming directly in machine language has understandably driven the development and increasing use of high-level programming languages; however, an environment where high-level language development dominates, can easily end up obscuring the link to machine language (even for those of us, who in their childhood may have been introduced to the concept – through computer platforms now long obsolete, such as Commodore 64 or ZX Spectrum). The most straight-forward mapping from machine instructions to a low-level language is preserved in assembly languages, where the portion of machine instructions specifying the operation of the CPU (known as opcode) is represented through alphabetic mnemonics, and the portion representing the operand data is written in a more human-readable form. Even with these alleviations, the tight binding to machine language makes assembly difficult to read and program in. However, a relatively straight-forward mapping to machine instructions can be obtained from other, more intuitive low-level languages, such as `C`.

A simple example of this can be seen on Fig. G.1 left (listing G.2:1a), which is a listing of a trivially simple program written in the `C` language, `min.c`, which adds two numbers and returns the result. Using a compiler (here, the GNU Compiler Collection or `gcc`), we can convert a source code program in `C` language to its executable, machine instruction, form – as an executable (or colloquially, "binary") file. Then, by using a disassembler, we can obtain an assembly language listing of the executable file; the disassembler used here (`objdump`) can, under certain settings, interleave the original `C` source code commands within the assembly listing. A portion of the assembly listing thus generated from the `min.c` executable is shown on Fig. G.1 right (listing G.2:1b); the first column stands for the virtual address of the machine instruction in hexadecimal notation, the second column contains the actual machine instruction rendered as hexadecimal bytes, and the third column shows the same machine instruction in an assembly format with a mnemonic. The lines on Fig. G.1 relate the locations of `C` commands in the original source code, with their positions in the assembly listing of the executable; and they demonstrate that

the sequential order of command execution as specified in the `C` code, has been preserved in the executable machine instruction form as well.

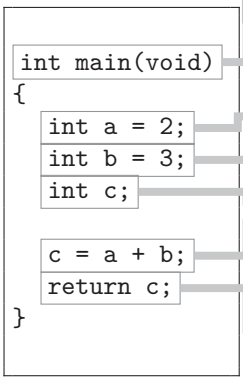
We can note that besides translating our main function (the entirety of the `min.c` code), the compiler has also automatically added sections related to initialization and termination of the program, which also link into the standard `C` library (under Linux, known as `libc`). It is also notable that we have to specify to the `gcc` compiler *not* to optimize code (through the command line switch `-O0`), in order to get such an obvious correspondence between `C` and assembly listings; otherwise (for optimization `-O3`, for example), the compiler would have detected that the two numbers being added, given they are hardcoded, will not change across different runs of the program – and so, the compiler would decide to simply return the value 5, as the entirety of the machine code translation of the `min.c` program!

Awareness of, simply speaking, a linear translation of a programming language to machine code (as shown on Fig. G.1), is not necessarily difficult for a programmer to gain, even with little previous experience with assembly language: essentially, it can be understood as sequential grouping of commands, similar to the case of subroutines (or functions). However, such a model of the translation process, may also imply that the machine instructions (and thus the originating `C` commands) are executed *immediately* one after another in time by the CPU – which turns out to be a gross approximation. Besides compiler optimization, which can significantly obscure the direct correspondence between `C` and machine code, facilities like threads and asynchronous event handling (available for the `C` language through libraries) introduce further difficulties in predicting the timing of execution of code: threads are supposed to give the impression of running in parallel (meaning, at the *same time*) as other function(s) on the same CPU; while asynchronous event handlers are functions that respond to events that, in essence, are expected to happen unpredictably randomly in time.

It becomes clear, then, that facilities which we've learned to expect on modern OSs, such as multitasking in a graphical user interface (GUI) desktop environment, would be difficult to implement - if different programs are to be *always* executed machine instruction by instruction, without any interruption, until they terminate. And indeed, that is why modern OSs make use of the concept of time (division) multiplex. This term in telecommunication refers to multiple signals transferred through a single channel by assigning them each a slice of time, where a particular signal would be exclusively present on the channel; similarly, in computing, it refers to an OS allocating each software *process* (corresponding to one [of many] instantiation of a program) a slice of time, where its machine instructions will be exclusively executed by the CPU. This, in turn, implies that the OS will also be in charge of pausing a process, once it has used up its time slice – and switching the execution to a different process for the next time slice; which implies that this part of the operating system must have a higher priority (or privilege) than any other software running on the system. This part of the OS is typically known as

Fig. G.1: Relating C and machine language code. Left: a simple program in `C` programming language; right: listing (edited) of machine instructions and assembly of the corresponding executable, generated by the compiler. The arrows indicate the translation of individual lines of `C` code to machine code.

Listing G.2:1a: Source code of `min.c`, compiled without optimization, using: `gcc -g -O0 min.c -o min.exe`



Listing G.2:1b: Machine instructions assembly of `min.exe`, obtained using:
`objdump -S min.exe`

```
Disassembly of section .init:

08048324 <_init>:
 8048324:  55                    push  %ebp
 8048325:  89 e5                 mov   %esp,%ebp
...
Disassembly of section .text:

080483c0 <_start>:
 80483c0:  31 ed                 xor   %ebp,%ebp
 80483c2:  5e                    pop   %esi
 80483c3:  89 e1                 mov   %esp,%ecx
...
 80483d7:  68 c4 84 04 08        push  $0x80484c4
 80483dc:  e8 a3 ff ff ff        call  <__libc_start_main@plt>
 80483e1:  f4                    hlt
 80483e2:  90                    nop
...

080484c4 <main>:

int main(void) {
 80484c4:  55                    push  %ebp
 80484c5:  89 e5                 mov   %esp,%ebp
 80484c7:  83 ec 10              sub   $0x10,%esp
 80484ca:  e8 d5 fe ff ff        call  80483a4 <mcount@plt>
  int a = 2;
 80484cf:  c7 45 fc 02 00 00 00  movl  $0x2,-0x4(%ebp)
  int b = 3;
 80484d6:  c7 45 f8 03 00 00 00  movl  $0x3,-0x8(%ebp)
  int c;
  c = a + b;
 80484dd:  8b 45 f8              mov   -0x8(%ebp),%eax
 80484e0:  8b 55 fc              mov   -0x4(%ebp),%edx
 80484e3:  8d 04 02              lea   (%edx,%eax,1),%eax
 80484e6:  89 45 f4              mov   %eax,-0xc(%ebp)
  return c;
 80484e9:  8b 45 f4              mov   -0xc(%ebp),%eax
}
 80484ec:  c9                    leave
 80484ed:  c3                    ret

...

Disassembly of section .fini:
080485cc <_fini>:
 80485cc:  55                    push  %ebp
 80485cd:  89 e5                 mov   %esp,%ebp
 80485cf:  53                    push  %ebx
 80485d0:  83 ec 04              sub   $0x4,%esp
 80485d3:  e8 00 00 00 00        call  80485d8 <_fini+0xc>
 80485d8:  5b                    pop   %ebx
 80485d9:  81 c3 1c 1a 00 00     add   $0x1a1c,%ebx
 80485df:  e8 5c fe ff ff        call  <__do_global_dtors_aux>
 80485e4:  59                    pop   %ecx
 80485e5:  5b                    pop   %ebx
 80485e6:  c9                    leave
 80485e7:  c3                    ret
```

the *kernel*; and indeed, time multiplexing (i.e. process switching) is one of the main tasks of the Linux kernel as well (among other tasks like memory and device management).

Conceptualizing this part of computer operation is made easier, if we think about what happens when one starts one too many programs on one's PC; however, it can be much more difficult to observe directly. Thankfully, the Linux kernel has a built-in debugging facility known as `ftrace`, allowing precisely this; an example is shown on Fig. G.2.

The lines on Fig. G.2 relate the first two machine instructions of the assembly listing from Fig. G.1, with their respective location in the logfile produced by `ftrace` (listing G.2:2b) – which, being timestamped, also determines more precisely *when* these instructions were executed. It is important to note that this kind of a log, internally called `function_graph`, traces only the entries and exits of Linux kernel functions; execution of user-space software, like the `min.c` program, is essentially transparent to it - therefore, additional measures have to be taken, so that execution of user-space instructions is also present in this log. In any case, the combination of user-space instructions, and the descriptive kernel function names (that may indicate the function's purpose even without a previous acquaintance with the kernel API) allow us to reconstruct how did that particular execution proceed, on the particular development OS.

The `ftrace` documentation notes that the system allows for choosing the source clock for timestamps: only local and global are present as choices on the 2.6.* series we used, although newer kernel releases offer more. The local clock is the default choice, and the source for listing G.2:2b as well; it is notable that the local clock, while fast, is not necessarily monotonic between CPUs. For instance, note that listing G.2:2b shows that the timestamp 1.685222 is reported twice: once for each CPU. In this case, it is impossible to determine which instruction got executed first in terms of an independent observer clock, even if the log formatting itself (as consecutive lines of text) automatically imposes some ordering; in a sense, this represents a real world illustration of the issues in distributed computing systems, discussed within the topic of Lamport timestamps [3].

From listing G.2:2b, we can see that the kernel in charge of process (or task) switching, indicated by functions like `pick_next_task_fair()` or `finish_task_switch()`, and by the task switching notices (`<idle> => min.exe`, `min.exe => kworker`). But beyond task switching, the kernel also manages the CPU migration: note that in this time slice, `min.exe` starts running on processor 1, but ends running on processor 0. This is a feature present on modern notebooks that use multiprocessor CPUs that address the same shared resources (like memory), also known as symmetric multiprocessing (SMP).

Furthermore, listing G.2:2b implies that, from the perspective of the kernel, the execution of a user-space software (like `min.exe`) simply consists of moving pages of memory (indicated by kernel functions like `handle_pte_fault()`, `do_wp_page()` and `vm_normal_page()`) to a given CPU, which the CPU then executes (without the kernel directly intervening in or "listening to" that). Ad-

Fig. G.2: Identifying machine code in a kernel log printout Left: snippet (edited) of executable machine instructions of `min.exe` (in mnemonic form only - same as in Fig. G.1 listing G.2:1b, but with byte representation of machine code omitted). Right: a kernel trace (edited), obtained during execution of `min.exe`. The columns on listing G.2:2b represent absolute time; CPU number and process name; and the function (or its exit) that executed at that point. The functions' indendation is relative to the particular function call stack, for the corresponding process and CPU.

Listing G.2:2a: Machine instructions of the `main()` function of `min.exe`

```
...
080484c4 <main>:

int main(void) {
 80484c4:  push %ebp
 80484c5:  mov  %esp,%ebp
 80484c7: sub  $0x10,%esp
 80484ca: call 80483a4 <mco
       ...
 80484cf: movl $0x2,-0x4(%ebp
       )
 80484d6: movl $0x3,-0x8(%ebp
       )
 80484dd: mov  -0x8(%ebp),%
       eax
 80484e0: mov  -0x4(%ebp),%
       edx
 80484e3: lea  (%edx,%eax
       ,1...
 80484e6: mov  %eax,-0xc(%ebp
       )
 80484e9: mov  -0xc(%ebp),%
       eax
}
...
```

Listing G.2:2b: A snippet of a function graph kernel trace, during execution of `min.exe`

```
 1.684497 | 1) <idle>-0 |   native_load_tls();
 1.684498 | 0)   bash   |         kmap_atomic_prot() {
 1.684499 | 0)   bash   |           native_set_pte();
 ------------------------------------------
 1)    <idle>-0    => min.exe
 ------------------------------------------
 1.684499 | 1) min.exe | schedule_tail() {
 1.684500 | 1) min.exe |   finish_task_switch();
 1.684501 | 0)   bash   |           arch_flush_lazy_mmu_mode();
 1.684502 | 0)   bash   |         }
 1.684503 | 1) min.exe | _  cond_resched();
 1.684503 | 0)   bash   |       }
 1.684504 | 0)   bash   | _      raw_spin_lock();
...

 1.685222 | 1) min.exe |   handle_pte_fault() {
 1.685222 | 0)   bash   | __                          slab_free();
 1.685223 | 1) min.exe | _    raw_spin_lock();
 1.685224 | 0)   bash   |                           }
 1.685225 | 1) min.exe |     do_wp_page() {
 1.685225 | 0)   bash   |                         }
 1.685226 | 1) min.exe |       vm_normal_page();
 1.685226 | 0)   bash   |                         file_free_rcu() {
...

 1.696925 | 0) rs:main |                                    jbd2_journal_add_journal
          ...
 1.696925 | 1) min.exe | /* 80484c4:  55 push %ebp */
 1.696927 | 0) rs:main |                                    do_get_write_access() {
 1.696927 | 1) min.exe | }
 1.696928 | 0) rs:main | _                                     cond_resched();
 1.696928 | 1) min.exe | native_set_debugreg();
 1.696930 | 0) rs:main |                                     unlock_buffer() {
 1.696931 | 0) rs:main |                                       wake_up_bit() {
 1.696932 | 0) rs:main |                                         bit_waitqueue();
 1.696932 | 1) min.exe | native_get_debugreg();
 1.696934 | 0) rs:main | __                                          wake_up_bit();
 1.696934 | 1) min.exe | native_set_debugreg();
 1.696936 | 0) rs:main |                                       }
 1.696936 | 1) min.exe | native_get_debugreg();
 1.696937 | 0) rs:main |                                     }
 1.696938 | 0) rs:main |                                     jbd2_journal_cancel_r
          ...
 1.696938 | 1) min.exe | native_set_debugreg();
 1.696939 | 0) rs:main |                                     }
 1.696940 | 1) min.exe | () {
 1.696941 | 0) rs:main |                                     jbd2_journal_put_journal
          ...
 1.696942 | 1) min.exe | /* 80484c5:  89 e5 mov %esp,%ebp */
 1.696943 | 0) rs:main |                                   }
 1.696943 | 1) min.exe | }
 1.696944 | 0) rs:main |                                 }
...

 1.708202 | 0) min.exe |         pick_next_task_fair() {
 1.708203 | 1)   bash  |             nsecs_to_jiffies();
 1.708203 | 0) min.exe |           clear_buddies.clone.58();
 1.708205 | 1)   bash  |           }
 1.708205 | 0) min.exe |           update_stats_wait_end.clone.93();
 1.708206 | 1)   bash  | _          raw_spin_lock_irq();
 1.708207 | 0) min.exe |           hrtick_start_fair();
 1.708209 | 1)   bash  | _          cond_resched();
 1.708209 | 0) min.exe |         }
 1.708212 | 1)   bash  |           release_task() {
 1.708212 | 0) min.exe |         native_read_cr0();
 1.708213 | 1)   bash  |             proc_flush_task() {
 1.708214 | 1)   bash  |               proc_flush_task_mnt.clone.5() {
 1.708214 | 0) min.exe |         native_write_cr0();
 1.708216 | 0) min.exe |         native_load_sp0();
 1.708218 | 0) min.exe |         native_load_tls();
 1.708219 | 1)   bash  |                 d_hash_and_lookup() {
 ------------------------------------------
 0) min.exe  =>  kworker-8516
 ------------------------------------------
```

ditionally, the kernel *always* has the priority to interrupt execution of a user program, even at the granularity level of a single machine instruction: as it can be seen, our first two machine instructions are separated by kernel calls to `native_set_debugreg()` within the same `min.exe` process. In this particular case, these calls are an artifact of our debugging setup; but the important thing in general is that the kernel – at *any* time – can interrupt execution of user-space programs. This shows that predicting the timing and order of execution of user (and kernel) programs, becomes increasingly difficult in an SMP environment (see also [4]).

The ability of the kernel to control the execution timing of user-space processes by switching them is known as process preemption, and it has been a feature of the Linux kernel for a long time [5]. However, the 2.6 series of kernels introduced the concept of *kernel preemption* - meaning that functions running as part of the kernel become preemptible themselves [6], [7]. In other words, with kernel preemption, the kernel can decide at any time to reschedule its own, high- priority functions – and this has a specific influence on the performance of kernel functions. In particular, here we're interested in those kernel functions, that we would expect would provide us with a consistently periodic clock tick, known as timer functions – which have been essential in our previous driver development work. This influence is the focus of the next section.

As a side note, let us mention that our example program, `min.c`, is so trivial, it will not produce any visible output when the executable `min.exe` is ran (we would have to separately print the return value from the terminal shell, after the program has terminated and returned, to see it). Anything else would have greatly increased the complexity of the example from the kernel perspective: printing the output value through a `C` API function like `printf()` depends on what the standard output is routed to; and in the most common use case of terminal emulators, may involve character device and terminal emulator driver calls to the kernel. On the other hand, writing that value to disk through a `C` API function like `write()` may be even more complex, as it includes block device, file system and hard disk driver calls to the kernel. The code used to generate the snippets on Fig. G.1 and Fig. G.2, as well as the original debug logs obtained for those figures, are available online via [1] under the name `trace-user-program`.

G.3 Standard vs. high-resolution timers in the Linux kernel

By programming a soundcard audio driver, we basically want to gain control, from the PC OS environment, of the analog-to-digital (ADC) and digital-to-analog (DAC) conversion processes occurring on the soundcard hardware. Since both ADC and DAC are defined as repeatable (or periodic) processes in time, driven by an oscillator clock - to address them, we similarly would need facilities that allow periodic executions of subroutines from software. Both `C` and higher

level scripting languages (like `Perl` or `Python`) have libraries available, which facilitate programming of functions that execute periodically in time.

It should be noted, that in general, we can think about periodically executing software code in several ways. For one, we can think of it as a thread, which executes its task code, and then sleeps for a specified interval of time, before looping and executing again. This is relatively easy to implement in the mentioned languages as user-space software; however, being aware of process preemption, it is clear that we cannot expect periodicity at *exactly* the sleep time: the kernel can (and will), at any time, reschedule the thread for execution later, and thus the only guarantee we have is that we will obtain periodicity with *at least* the sleep time as period, but most likely *more* by a random amount of time. In other words, we cannot expect $t_a = T_s$; only that $t_a \geq T_s$, where t_a is actual time of execution of a code loop, and T_s represents the requested sleep time as a period.

In terms of hardware, this means that for: a hardware clock running on the soundcard hardware; and a periodic PC software subroutine as described; which are started at the same time - the periodic subroutine will necessarily demonstrate clock *drift* in respect to the oscillator clock on the peripheral hardware, as it runs later than it on average. Thus, even if a certain timing uncertainty in the subroutine period would be introduced just by the very existence of conditional branching in the subroutine's task code, it is reasonable to attempt to minimize the effects of user-space process preemption when addressing hardware. Thus, soundcard (and other) device drivers would naturally belong to kernel space, assuming the form of kernel modules - and it is here, where our primary interest in an appropriate API for periodic subroutines lies.

Under Linux, the primary - or for the purposes of this document, standard - kernel `C` language API to allow for periodic subroutines, is formed by the `struct timer_list` and the functions `init_timer()`, `add_timer()`, and `del_timer()`, provided by the header `<linux/time.h>`. While this API receives but a skint mention in /Documentation/DocBook/kernel-locking.tmpl and /Documentation/local_ops.txt files of the documentation, that follows with the kernel source code for this series – it is described in more detail in [8, Chapter 7, "Time, Delays, and Deferred Work"] as "The Timer API". The same API is also used in our virtual `minivosc` [2a], as well as the `AudioArduino` [3a] ALSA drivers. It is notable that in this API, the `add_timer` function is, effectively, a *one-shot*: we specify a timer function to be executed, and time when it should be executed, and then we call `add_timer()` to start the process; when that time arrives, the function is executed - and will not be executed again, unless we call `add_timer` (or alternatively, `mod_timer`) again from within the function. Periodic repetition is thus achieved, through the timer function repeatedly scheduling itself, at a constant interval in the future – that is, at a time which is a constant duration away in the future from the current one.

An important thing to realize about this API, is that within it, time is expressed at a resolution of a *jiffy*. The duration of one jiffy is platform dependent, and it depends on the kernel configuration variable `HZ`, which is set

at kernel compile time. This `HZ` value is used to program the timing hardware of a PC at kernel boot time, which will thereafter generate a timing interrupt, that serves as the main system *tick* of the kernel. This tick "*is the CPU's cue to reconsider which process should be running, catch up with read-copy-update (RCU) callbacks, and generally handle any necessary housekeeping* [9]". At each such tick, the internal kernel variable `jiffies` is increased, and as such "*it represents the number of clock ticks since last boot* [8]". The duration of one jiffy (T_j) would then be simply:

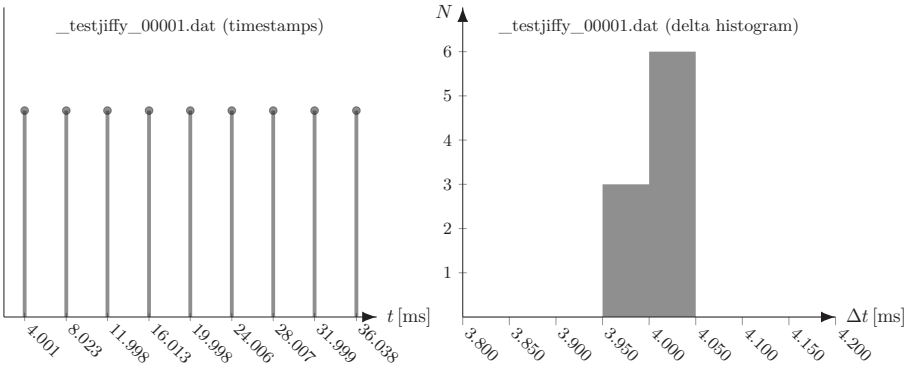
$$T_j = \frac{1}{\texttt{HZ}} \tag{G.3.1}$$

The value of `HZ` is typically between 100 and 1000, and can be obtained by printing it from a kernel module, or from the kernel build configuration file; the kernel can make it available through the `/proc` filesystem, but it doesn't do so for our development platform. On our machines, the kernel configuration is provided as a file named `/boot/config-*`, with the kernel release string appended. The problem is that for our 2.6.38 machine, it reports both `CONFIG_HZ=250` and `CONFIG_NO_HZ=y`. The `CONFIG_NO_HZ` option implies a tickless kernel, described in /Documentation/timers/NO_HZ.txt (in kernel sources): essentially, it will turn off the system tick if the CPU is idle for a long period, for power management purposes. The value can also be double-checked by utilizing the kernel-managed file `/proc/timer_list` and for our 2.6.38 machine it returns 250; thus the duration of a jiffy on this platform, as per Eq. G.3.1, would be 4 ms.

The most straightforward way to debug the Linux kernel is by using the `printk()` kernel function: it will output a supplied message to the system log (on our platforms, represented by the file `/var/log/syslog`), prefixed by a timestamp, formatted as a floating point number with 6 decimals (just like the timestamps on the `ftrace` log in listing G.2:2b) which represents seconds. While this may imply microsecond resolution of the `printk()` timestamp, as a rule of thumb it should be assumed that there is *no* guarantee of that: the timestamp obtained may have a larger granularity than 1 μs. Thus, one way to observe a periodic timer function with a period of one jiffy, is to write a kernel module which starts a timer function with the standard Timer API, whose only task would be to print a message to the log, and then reschedule itself to 1 jiffy in the future. We have written a small kernel module named `testjiffy`, which does exactly that - its timer function is started when the module is loaded, and automatically stops after 10 iterations; timestamps from the messages can be extracted by post-processing the log, and graphed using a plotting application like `gnuplot`. We have written a small script to automate this process, and result of one such run of the script is shown on Fig. G.3.

The sequence shown on Fig. G.3 left, does correspond to the expected behavior of this driver: all of the timestamps of the kernel log messages, produced by the module's timer function, indeed are about 4 ms – the duration of a jiffy – apart, thus forming a periodic sequence. However, there is also *jitter*: small enough that it is not visible on the time sequence plot, but visible both on the

Fig. G.3: Expected behavior of the standard Linux timer in the `testjiffy` module. The left-hand side plot is in the time-domain, and shows the positions of the timestamps (of system log messages); the sequence is offset so the first timestamp (not plotted) starts at 0. The right-hand side plot is an absolute count histogram of time intervals (deltas) between consecutive timestamps, with bin width 0.05 ms; average (n=9) is 4004.200 μs (σ = 18.407 μs)

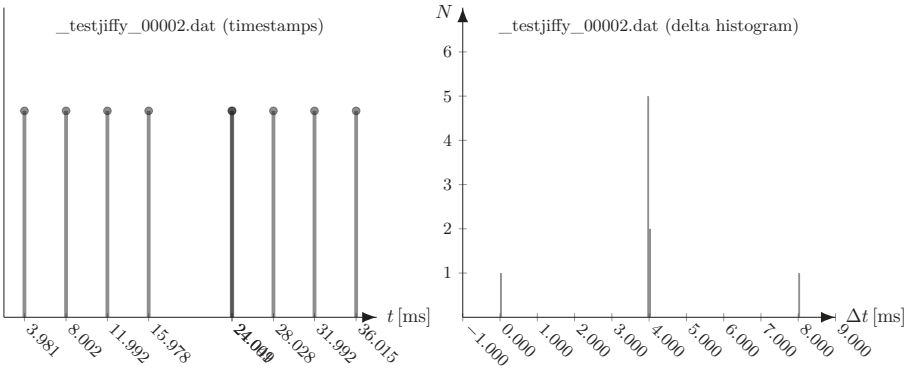


consecutive timestamp delta histogram on Fig. G.3 right – and as a standard deviation of σ=18.407 μs. The interesting thing here is that we have obtained time deltas that can be either larger or smaller than the reference jiffy period of 4 ms.

The larger deltas we could easily attribute to kernel preemption; however the smaller ones are difficult to interpret otherwise, than as an existence of an algorithm that attempts to minimize the effects of the overall clock drift. While these reasons are simply a hypothesis (whose confirmation through a dedicated debug procedure is outside the scope of this project), the existence of the jitter is a clearly visible fact. It is notable that this jitter happens, because the kernel "*executes timers in bottom-half context, as softirqs, after the timer interrupt completes* [10]" – and a software interrupt (or softirq) represents the second highest level of priority in the Linux kernel; right below hardware interrupts. However, while the situation on Fig. G.3 reappears on most runs of the test on our development platform, it does not occur always. In fact, relatively often, a somewhat significant departure from the expected can occur; a log captured from one such run is plotted on Fig. G.4.

This time, Fig. G.4 (left) quite explicitly shows that one of the expected impulses is "dropped". A closer inspection reveals that it is not missing, but after the gap, there are two impulses: logged as only 48 μs apart, on the sequence plot they appear to overlap (emphasized by the stronger pulse color, and the axis labels overlap). Simultaneously, the histogram on Fig. G.4 right shows occurrences of a delta, one close to 0, and another one close to twice the jiffy period (while the others cluster around the jiffy period like on Fig. G.3). This implies that the execution of the timer function was not only preempted, but

Fig. G.4: Behavior of the standard Linux timer in the `testjiffy` module exhibiting a rescheduling "drop". The left-hand side plot is in the time-domain, and shows the positions of the timestamps (of system log messages); the sequence is offset so the first timestamp (not plotted) starts at 0. The right-hand side plot is an absolute count histogram of time intervals (deltas) between consecutive timestamps, with bin width 0.05 ms; average (n=9) is 4001.66 μs (σ = 1.879 ms)



also rescheduled: since the kernel has no smaller option for rescheduling in the future, than increasing the expiration counter of the timer (expressed in jiffies) by 1 – effectively, this would result with rescheduling the timer function at the next system tick. However, at that time (say, at i jiffies), the kernel would also have scheduled the regular, "next expected" timer function iteration, to occur at $i + 1$ jiffies. Thus, both the re-scheduled, and the normally scheduled, iterations of the timer function would be set to run at $i + 1$ jiffies – which would account for the "overlapping" impulses, which record the timer functions running very shortly after each other in quick succession.

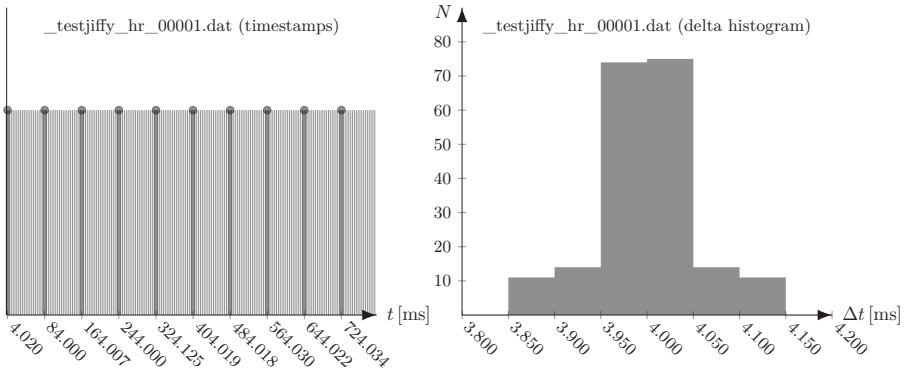
While confirming the exact sequence of events leading to the behavior shown on Fig. G.4, is again a debugging effort beyond the scope of this project - the important thing is that, in terms of periodic functions, this demonstrates jitter (or in general, timing *uncertainty*) of up to the entire expected period duration, if we aim at the smallest possible period with this API (the duration of a jiffy). This is significantly higher than a case like on Fig. G.3, where the uncertainty, expressed through standard deviation, is two orders of magnitude smaller than the period duration itself. We should also note that quite often, we have also obtained captures with a timing gap like on Fig. G.4, but *without* two "overlapping" impulses – which would represent an actual case of a timer function being "dropped" (with the meaning of "left out"), as the plot in that case, contains one less impulse than the expected 9.

At this point, we should mention that due to the small number of iterations, the statistics on Fig. G.3 and Fig. G.4 is not meant to be an indicator of general behavior; instead, it demonstrates that kernel timer function uncertainty of up

to a jiffy is relatively easy to stumble upon and record - even in circumstances of otherwise no significant OS load. In all, this shows that the standard kernel Timer API is unreliable as a clock mechanism, for a periodic function with the minimum possible period of one jiffy – however, we should add that this is not a problem if the repetition period increases: at a period of, say, 400 ms, an uncertainty of 4 ms already starts becoming insignificant, and grows more so as the period increases.

Specifically, we can see this as a timing resolution problem: the standard kernel Timer API simply does not have a mechanism, that can manage time duration less than a a jiffy, and can only manage integer multiples of this quantum. However, there exists a mechanism in this kernel release, which addresses precisely that - known as the high-resolution kernel timer API, often called "hrtimers". We have written a small kernel module named `testjiffy_hr`, which uses this API to do exactly the same as `testjiffy` did previously: run a timer function (which simply writes a timestamped message in the system log) repeatedly with a period of one jiffy (4 ms), and stop it after a certain amount of repetitions (here 200). The analysis of the message timestamps is otherwise the same: we plot the timestamps in the time domain, and their deltas as histogram – and a capture of one such typical run is shown on Fig. G.5.

Fig. G.5: Behavior of the high-resolution Linux timer in the `testjiffy_hr` module. The left-hand side plot is in the time-domain, and shows the positions of the timestamps (of system log messages); the sequence is offset so the first timestamp (not plotted) starts at 0. Only each 20th timestamp is emphasized, to give a sense of scale; the remaining timestamps are represented by a thin line. The right-hand side plot is an absolute count histogram of time intervals (deltas) between consecutive timestamps, with bin width 0.05 ms ; average (n=199) is 4000.110 μs (σ = 53.382 μs)



While the hrtimers kernel API is not mentioned in, say, [8], its design is addressed in both /Documentation/timers/hrtimers.txt and /Documentation/-timers/highres.txt in the kernel source tree, as well as in [11]. It is a relatively recent addition to the kernel, present since the 2.6.16 release [12]. Even if it is difficult to observe the detail of time sequence on Fig. G.5 left, by subsampling

at a regular interval (like emphasizing each 20th sample), we can perceive at first glance a relatively stable period on this scale. The histogram on Fig. G.5 right reveals more detail: while the standard deviation is somewhat (nearly 3 times) larger than the one on Fig. G.3, the majority of the time deltas on Fig. G.5 fall in the same two bins as on Fig. G.3. The histogram is also quite symmetrically centered around the target period of 4 ms; we can again surmise this as evidence of a kernel algorithm aiming to compensate errors that have resulted with deltas longer than the period, by scheduling other timer function calls at deltas shorter than the period. In other words, once a timer function starts executing later than the expected period, is already *too late* to do anything about it; we *cannot* go *back in time*, and force the kernel to execute the timer function at the otherwise appropriate moment in the past. The kernel can only try to compensate for the effects of this error, by scheduling the next timer function a little early: if it executes as planned, it would then bring the periodic execution back in sync, in principle at least.

We should note that in our experience with repeated test runs of `testjiffy_hr` under conditions of no significant OS load, we have never observed a "drop" or a reschedule on the order of the 4 ms jiffy period as on Fig. G.4: most of the time, the distribution of time deltas remains similar to Fig. G.5 (where the span of active bins represents but 7.5 % of the 4 ms period); although, the distribution is influenced by increasing OS load. As such, we find the hrtimers API to be a reliable clock mechanism, for implementation of a periodic function, with a period on the order of 4 ms and longer (and as seen later in subsection G.6.3, Fig. G.32, also shorter).

The high-resolution timers' `C` language kernel API, allowing for periodic subroutines, is formed by the `struct hrtimer` and the functions `hrtimer_init()`, `hrtimer_start()`, and `hrtimer_cancel()`, provided by the header `<linux/hrtimer.h>`. In contrast to the standard kernel Timer API, here we don't reschedule the timer function "from itself" anymore; instead, the period duration and the `HRTIMER_MODE_REL` constant are supplied to the `hrtimer_start()` call, and in return, the timer function is repeatedly rescheduled, as long as it returns the constant `HRTIMER_RESTART`; the periodic behavior can be terminated by returning the constant `HRTIMER_NORESTART` from the timer function instead. This kind of organization seemingly allows for the hrtimers API to be "informed" about the demand for periodic performance; and as such, may have better opportunities to compensate for random preemption drift – which might clarify why the histogram on Fig. G.5 right, seems rather balanced.

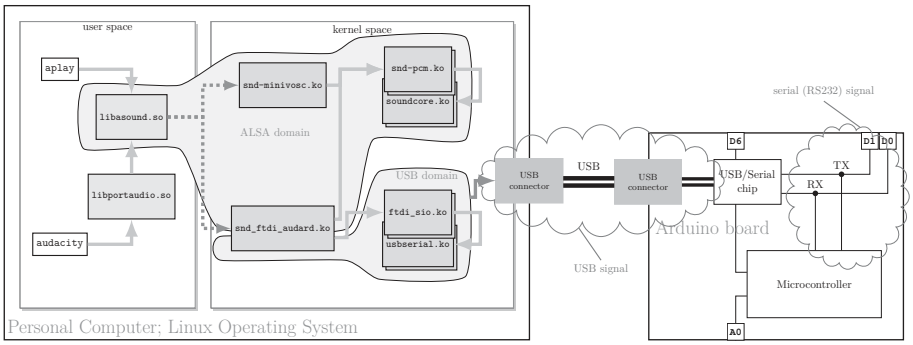
The code for the `testjiffy` and `testjiffy_hr` modules, as well as scripts and original debug log data used to generate Fig. G.3, Fig. G.4, and Fig. G.5, are available online via [1] under the name `testjiffy`. Developing these modules was impeded by our previous experience, as we have already used the standard Timer API, at a period of one jiffy, *previously*: both in a virtual [2a] and a real [3a, 5a] soundcard driver context; and yet, we have *never* observed any problems with their performance, that would point to periodic timer functions being postponed for an entire period duration. And in light of the discussion

so far, it is difficult to accept that such delays or drops were not happening; it is far more likely that they indeed were occurring, but their effects ended up being masked by the specific operation regimes of those drivers. This kind of outcome is examined in more detail in the next, section G.4.

G.4 The effect of period-long timer function jitter, with streaming data rates as parameter

Let us first consider the context of Minivosc (with kernel module `snd-minivosc.ko` [2a]) and AudioArduino (`snd_ftdi_audard.ko` [3a, 5a]) ALSA drivers, shown on Fig. G.6.

Fig. G.6: A block diagram overview of the context of some of our development. The PC OS block visualizes executable programs and shared libraries (`.so`) as part of the user space; and specific kernel modules (`.ko`) as part of the kernel space. An Arduino board, representing a soundcard peripheral device, is connected via USB to the PC; a USB/Serial chip on the board converts the USB signal to RS232 serial signal (carried through by RX and TX [receive and transmit, from perspective of the microcontroller] lines), which is what the microcontroller on the Arduino communicates through. The microcontroller, as a soundcard in AudioArduino, handles analog I/O through its analog (`A0`) and digital (`D6`) pins.



In user-space, ALSA is represented by the shared library object `libasound.so`; audio applications like `aplay` or `audacity` are ultimately linked against this library, so they can use the ALSA API functions. However, note that `audacity` uses the PortAudio library, which provides a consistent user-space cross-platform API for soundcard operation; as such, `audacity` is actually primarily linked against `libportaudio.so`; which then ensures proper hooking into `libasound.so`. As such, PortAudio represents an extra layer of indirection into the audio subsystem of Linux. Note that on our development OSs, there is an additional audio library, PulseAudio [13], [14], activated by default; we keep it turned off, to avoid handling this additional layer of indirection during debugging.

Depending on the choice of soundcard made in the application, `libasound.so` then utilizes the right soundcard driver, here either `snd-minivosc.ko` or `snd_`

`ftdi_audard.ko`. In kernel space, ALSA can be said to be a software solution stack, because reusable parts of the functionality are organized in separate kernel modules like `soundcore.ko`, `snd-pcm.ko` and others; thus the "soundcard driver" is the kernel module that only deals with actions that are specific to each soundcard hardware device. So, both of our drivers work in concert with the rest of the ALSA modules, and this could be said to represent the ALSA domain. Since `snd-minivosc.ko` is a virtual driver, it does not need to interact with any other kernel subsystem; however, `snd_ftdi_audard.ko`, as a re-implementation of the default `ftdi_sio.ko`, is also a USB driver – and as such utilizes functions in other modules of the Linux USB stack, like `usbserial.ko` and others.

When we have hardware operation through the `snd_ftdi_audard.ko` driver, the PC communicates with the USB/Serial chip on board the Arduino through a USB signal. The USB/Serial chip on our (now vintage) Arduino Duemilanove is FT232RL from FTDI (which is no longer present on current editions of the Arduino board). We usually try to abstract, to the greatest extent possible, focusing on the USB signal in our projects, because its analysis involves the use of the still relatively expensive USB analyzer instruments. The USB/Serial chip can be said to translate the USB signal to RS232 (TTL levels) serial signals, which are used for communication with the Atmel ATmega328 microcontroller (documented in the 660-page datasheet [15]) on board the Arduino. The serial communication parameters are set to 8N1 (8 data bits, no parity, 1 stop bit) at 2 MBd (2 million Baud); and this serial traffic can be taken to represent the bottleneck of the data transfer rate of the system in first approximation. The RS232 signal voltage can be relatively easily observed on an oscilloscope with the appropriate bandwidth; however, analysis in this context would require that we can record longer snippets of multiple binary signal voltages, which is the domain of digital logic analyzers. In recent times, relatively low-price logic analyzers have appeared, that can capture and decode serial traffic with our parameters; and for this project we used a Saleae Logic analyzer to capture RX and TX signals' data (see Fig. G.25). With analyzer captured data, we'd have a measure of the data communication rate and behavior in hardware - independent of the impression the software driver parameters (as well as software measurements of driver performance at those parameters), on their own, would give.

There exist contexts where 1 "Baud" [Bd] as a unit is equivalent to 1 "bit per second" [bit/s]; however, that is not the case if we use 8N1 RS232 serial communication. 8N1 imposes that transmitted data is formatted as 1 start bit, followed by 8 data bits, followed by 1 stop bit; as such, there are a total of 10 signal (or symbol) transitions for each byte - for each 8 bits of actual data we want to transmit. Let us parametrize this mathematically, with illustration of some of these parameters on Figures G.7, G.8, and G.9. The baud rate (f_{Bd}), in this case, more properly stands for "signal transitions per second" – and based on it, we can derive a "baud period" (T_{Bd}) as the duration of time each

symbol is allocated on the line:

$$T_{Bd} = \frac{1}{f_{Bd}} \tag{G.4.1}$$

Let N_{stB} be the number of signal transitions per packet - in this case, the packet is a byte, and N_{stB} is 10, counting the start and stop bits. Then, the "byte period" (T_B) would be the time required to transmit 1 Byte (or 8 bits) of information:

$$T_B = N_{stB} \cdot T_{Bd} \tag{G.4.2}$$

Let N_{dbB} be the number of actual data bits per packet - the packet here being a byte, it is clear that N_{dbB} is 8. The effective "data bit period" (T_{db}) would then be obtained from the byte period T_B:

$$T_{db} = \frac{T_B}{N_{dbB}} \tag{G.4.3}$$

We can now express an effective serial byte rate f_B from Eq. G.4.2, in "bytes per second" [B/s], as:

$$f_B = \frac{1}{T_B} = \frac{f_{Bd}}{N_{stB}} \tag{G.4.4}$$

... and an effective serial data bit rate f_{db} from Eq. G.4.3, in [bit/s], as:

$$f_{db} = \frac{1}{T_{db}} = f_{Bd} \cdot \frac{N_{dbB}}{N_{stB}} \tag{G.4.5}$$

Thus, for our serial traffic specification of 8N1 at baud rate $f_{Bd} = 2\,\text{MBd}$, we have effective data bit rate of $f_{db} = {}^{8}/_{10} \cdot 2{\cdot}10^6 = 1.6\,\text{Mbit/s}$, and an effective data byte rate of $f_B = {}^{1}/_{10} \cdot 2{\cdot}10^6 = 200\,\text{kB/s}$ (with byte period $T_B = 5\,\mu\text{s}$). This would be the ultimate (idealized) limitation of the data transfer rates in this setup, per direction – in our experiments related to [3a], we have produced code, that on our development platform can demonstrate typically around 98 % of this maximum, in either direction in full-duplex mode (as measured solely from the perspective of that user-space code.)

We have surmised that this maximum data rate is the primary means for determining the viability of a soundcard driver on a given platform. For example, the 8 bit/44.1 kHz/mono settings of our `snd_ftdi_audard.ko` driver impose an audio byte rate $f_{aB} = 44.1\,\text{kB/s}$; and a corresponding "audio byte period", as the time duration required to transmit a byte, as $T_{aB} = 22.67\,\mu\text{s}$:

$$T_{aB} = \frac{1}{f_{aB}} \tag{G.4.6}$$

Since the audio data will eventually have to be formatted in serial 8N1 format, it has to be satisfied that $f_{aB} < f_B$: the audio byte rate has to be smaller than the serial byte rate; which holds in this case. However, for more accurate description of the behavior in the time domain, we have to take into

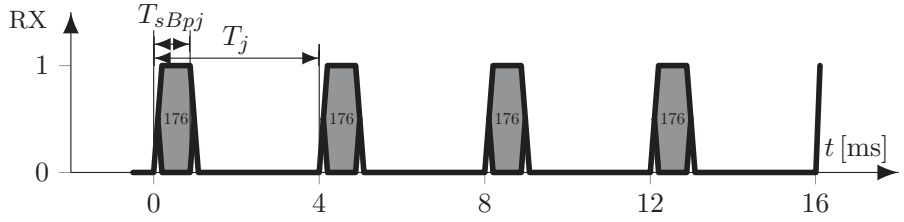
account the driver architecture. The `snd_ftdi_audard.ko` driver starts a timer function from the kernel standard Timer API, at a period of one jiffy, $T_j = 4\,\text{ms}$. The number of bytes that should be transmitted in that period of time, as per the audio byte rate, would be N_{aBpj}:

$$N_{aBpj} = \frac{T_j}{T_{aB}} = T_j \cdot f_{aB} \tag{G.4.7}$$

In this case, $N_{aBpj} = 176.4$; however, since we can only send an integer amount of bytes through a function call, in practice this means that if we settle for a rounded down value of $N_{aBpj} = 176$, we would be implementing a slightly slower audio transfer byte rate (here, 44 kB/s) than the required one. This means that the timer function should check past performance, and increase the number of bytes transmitted per call at opportune times, to compensate for the rounding-off error; the `snd_ftdi_audard.ko` driver doesn't do this - but interestingly, we have not observed any problems in our tests of it. In any case, each jiffy period T_j we transmit N_{aBpj} bytes – however, these bytes end up on wire in the serial domain, where each of them takes time T_B to transmit. Therefore, the time it takes to transmit the entire packet of N_{aBpj} bytes through serial would be:

$$T_{sBpj} = N_{aBpj} \cdot T_B = \frac{N_{aBpj}}{f_B} \tag{G.4.8}$$

Fig. G.7: Idealized serial traffic response to a periodic timer function with period $T_j = 4\,\text{ms}$, initiating transmission of $N_{aBpj} = 176$ bytes per call (equivalent to $f_{aB} = 44.1\,\text{kB/s}$), over a serial line with byte rate of $f_B = 200\,\text{kB/s}$. The idle state between the packets is visualized with a low logic level.



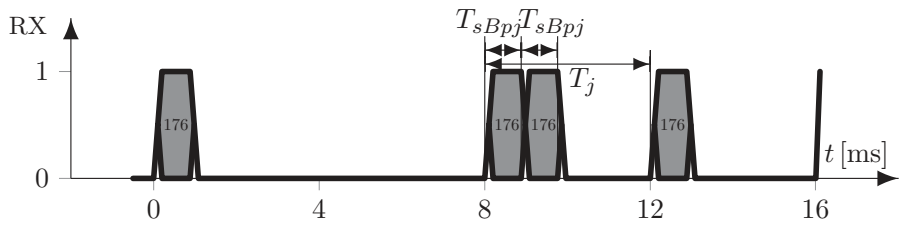
In this case, for $N_{aBpj} = 176$, T_{sBpj} is 880 μs - and it represents 22 % of the jiffy period $T_j = 4\,\text{ms}$. This is visualized on the Fig. G.7.

The response on Fig. G.7 is idealized, because it approximates away any jitter that would be inherent in this setup. For the playback direction, the timer function of `snd_ftdi_audard.ko` calls the function `ftdi_write()`, which then schedules a request (a URB - USB Request Block, discussed further in sect. G.6.1), and returns immediately – and it is up to the kernel to decide *when* it will honor the request, and actually activate the sending through USB. This is a layer of uncertainty, *additional* to the jitter inherent in the execution

of timer functions, as seen through the time of entry into the timer function itself. Furthermore, it assumes that when the USB signal from the PC gets converted to serial, the bytes would be reproduced on the RX line "back to back" – and while there is also jitter inherent in this stage, in our experience that approximation is not too far off. With that in mind, let's consider what would happen on wire, when the kernel goes through a rescheduling "drop" of the timer function on the order of a jiffy period, like on Fig. G.4.

One consequence of this behavior of `ftdi_write()`, is that the USB stack acts like a first buffer for the data we write. Thus, when the timer function is delayed like on Fig. G.4, and we get two quick consecutive runs of the timer function a period later, this results simply with quick scheduling of more data to send for the USB stack. Then, we can assume the USB stack will send the data, honoring the data rate limitation set by the serial traffic parameters – resulting with the situation shown on Fig. G.8.

Fig. G.8: Idealized serial traffic response to a periodic timer function, which exhibits a rescheduling "drop". The timer function did not execute at time T_j, being rescheduled for $2T_j$ – at which time, it executes very close to the timer function already scheduled for that time slot; this queues twice the expected amount of bytes for transmission.



Recall that for the reschedule drop on Fig. G.4, we noted that the quick succession of the two timer functions are but 48 μs apart; here the timer function is a bit bigger, but we could still take that as a rule of thumb, the combined execution duration of the quick succession of timer functions will be on the order of hundreds of microseconds. These will just schedule data for sending, and assuming that the buffered data will be sent back to back, the time to send all this data will now be $2T_{sBpj} = 1.76\,\text{ms}$. Since the timer functions schedule data, thus filling the buffer, quickly; and it takes longer time than that to empty the buffer, by sending the data on wire – this situation can be seen as an example of the concept of *leaky bucket* in computing (see e.g. [16]).

Thus, even if the timer function experiences an error in terms of periodic scheduling - Fig. G.8 shows us that, by the 4^{th} call (packet at 12 ms), the stream synchronization has recovered; and is back to having sent the same amount of data, as in the reference case on Fig. G.7. It is clear from Fig. G.7 that such a recovery is only possible as long as $2T_{sBpj} < T_j$, or written otherwise:

$$T_{sBpj} < \frac{1}{2} \cdot T_j \tag{G.4.9}$$

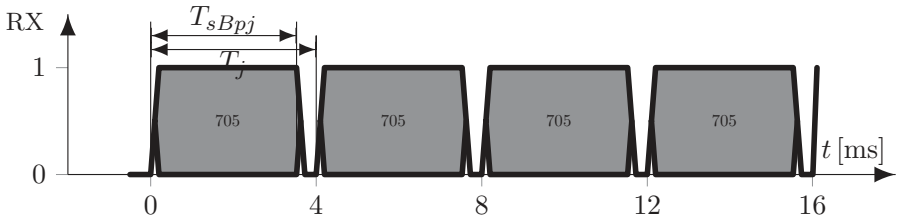
Reformulating this expression (G.4.9) in terms of our a priori settings: the audio byte rate targeted by the audio driver f_{aB}; and the hardware serial byte rate imposed by the serial traffic settings f_B; we obtain:

$$f_{aB} < \frac{1}{2} \cdot f_B \tag{G.4.10}$$

Interestingly, while we started by searching for a criteria, for a recoverable rate in case of a jitter error (reschedule) on the order of the period of the timer function T_j – that parameter (and correspondingly, `HZ`) got cancelled out, and we're left with expression (G.4.10), which simply states that the audio byte rate should be less than half of the effective serial byte rate (which is the hardware bottleneck here).

Now, let's see what will happen if we simply change the original settings of our `snd_ftdi_audard.ko` driver from 8 bit/44.1 kHz/mono to CD-quality (16 bit/44.1 kHz/stereo). This increases the audio byte rate fourfold, so $f_{aB} = 176.4\,\text{kB/s}$. Comparing it to the serial byte rate $f_B = 200\,\text{kB/s}$, we can see that inequality G.4.10 does not hold any more. Considering that in this case, N_{aBpj} as per Eq. G.4.7 would be $705.6 \approx 705$, and as per Eq. G.4.8, $T_{sBpj} \approx 3.525\,\text{ms}$, it is not difficult to see why on Fig. G.9:

Fig. G.9: Idealized serial traffic response to a periodic timer function with period $T_j = 4\,\text{ms}$ initiating transmission of $N_{aBpj} = 705$ bytes per call (equivalent to $f_{aB} = 176.4\,\text{kB/s}$), over a serial line with byte rate of $f_B = 200\,\text{kB/s}$.



Namely, we now have a leeway $T_j - T_{sBpj}$ of only some 475 μs – only 11.8 % of the timer function period – for jitter errors; any larger jitter than this, and we start filling the leaky bucket faster that it can get emptied, which is bound to eventually result with errors in operation. This could explain why we never observed problems in operation with the 8 bit/44.1 kHz/mono version of `snd_ftdi_audard.ko`, even if it is pretty certain that such reschedules were happening – and why we immediately started perceiving problems like dropped data, once we increased the data rate demands to 16 bit/44.1 kHz/stereo, while leaving the rest of the standard Timer API usage in the driver unchanged.

But what about the high-resolution timer API? For the particular measurement on Fig. G.5, we have a standard deviation of 53.382 μs, and the covered bins' range implies a maximum deviation from the period no larger than 150 μs, which is about three times smaller than the leeway of 475 μs. This would imply, at first glance, that the high-resolution timer would be appropriate even

for the higher, CD-quality data rate as on Fig. G.9. However, Fig. G.5 is but one measurement, and the deviations from the timer period will also depend on how many other applications are running at the time in the background – the OS load. As such, it is not difficult to imagine that for increased OS load, the deviations could easily break the leeway boundary, again resulting with problems due to the inability to exhaust the leaky bucket data. Finally, while the kernel does attempt to compensate for late scheduling errors, it does not necessarily succeed in doing so in equal measure; in which case, we should assume that the jittering errors will be cumulative, causing clock drift that breaks the audio rate synchronization of the data stream. So, while with the hrtimer API (as opposed to the standard timer API) we can expect jitter less than a timer period (or even less than the leeway boundary) under at least some conditions, making it appropriate for higher data rates – we cannot expect that its usage alone will address the entirety of the influences on the uncertainty, otherwise present in a preemptive kernel system.

G.4.1 Visualizing and sonification of timestamped log files with numStepCsvLogVis

At this point, let's make a slight digression, and take a look at the process that led us to the conclusions in Sect.G.2, Sect. G.3 and Sect.G.4. Not having the understanding outlined in that discussion, we were led to inspecting other reasons for the misbehavior of our drivers (that turned out to be unrelated), among them the wrapping behavior of circular buffers. The primary tool for inspection is printing debug messages in a log, timestamped similarly to listing G.2:2b, from various points in user- or kernel-space code – except we also print numeric values of specific variables (like the head or tail value of a circular buffer). The problem is: not knowing a priori what one should look for, one tends to arbitrarily add variables to a debug message; so in the end, we may end up with something like on listing G.4.1:1.

Listing G.4.1:1: A snippet of syslog messages containing timestamps and numeric values of variables

```
...
May 26 12:01:14 mypc testTimeSB[23879]: [134868.424837] start 8-3 end: 7,2 size: 11 ; value
      18 3
May 26 12:01:14 mypc testTimeSB[23879]: [134868.425030] start 9-4 end: 8,3 size: 11 ; value
      19 4
May 26 12:01:14 mypc testTimeSB[23879]: [134868.425216] start 10-0 end: 9,4 size: 11 ;
     value 20 0
May 26 12:01:14 mypc testTimeSB[23879]: [134868.425403] start 0-0 end: 10,0 size: 11 ;
     value 21 1
May 26 12:01:14 mypc testTimeSB[23879]: [134868.425591] start 1-1 end: 0,0 size: 11 ; value
      22 2
...
```

Listing G.4.1:1 shows a typical snippet: there may be additional data, irrelevant for our analysis, representing local time, PC name, process name and

id (added automatically to to `/var/log/syslog` printouts) - which is, however, not a static prefix (e.g. the local time will change), and as such, requires a regular expression to clean up. The actual (microsecond resolution) timestamp we're interested in, is in the next column, surrounded by square brackets; followed by the debug message, which is not consistently formatted at all: variable names can be followed by one or two values; names and values are separated by either whitespace or interpunction. Also, numeric values change the number of decimals they are written with; which, viewed in fixed width / monospaced / typewriter font (as used by default in terminal emulator software), breaks any implied column alignment. Additionally, at different points in the code, different variables exist - so formatting consistency of debug messages, across different points in the code, becomes a challenge as well.

It should be, therefore, no surprise, that any significant effort (beyond a cursory glance) in reading the debug log directly, presents a significant cognitive load: we might be interested in whether the transition of the first `start` value from 10 to 0, and the first `end` value from 9 to 10, is expected for the transition from time 134868.425216 to 134868.425403 on listing G.4.1:1; but that requires not only doing three subtractions in one's head – but also parsing the messages, so superfluous words and possible in-between lines are ignored. Of course, it is possible to take measures, and ensure greater formatting consistency of debug log messages – however, that implies that one already knows what one looks for. In a development context like ours, where debugging did not result with conclusive data in weeks, which caused changes to code and debug message format by the day – the additional effort required for ensuring formatting consistency of debug messages, quickly started to feel as an irrelevant and unacceptable trade-off.

Thus, we abandoned efforts to format debug log messages for direct reading, and opted instead for visualization, primarily with `gnuplot`. However, we found it difficult to use when we wanted to "zoom into" and analyze a small region of interest in a large dataset. To address this, we have produced a collection of Python scripts, called `numStepCsvLogVis`. Among them is `numLogfile2Csv.py`, which represents a generic parser, which will attempt to extract name/value data as on listing G.4.1:1, from differently formatted lines, automatically – and then output all numeric values as a comma-separated values (CSV) formatted text file, where the first line contains the automatically deduced variable names. Of course, there are limits to how general such a parser can be - for instance, without further information, it is impossible to tell if the `"-"` in `"10-0"` on listing G.4.1:1 should represent an algebraic "minus", or merely a separator sign; but it was usable to us on several occasions. For example, it turns listing G.4.1:1 into listing G.4.1:2.

The main part, and the namesake, of the collection is `numStepCsvLogVis.py` – an interactive application, whose screenshot is shown on Fig. G.10.

The application can read CSV data from a file (or piped from the standard input), and thereafter offers several modes of interaction, most of it happening through the terminal; and as such, in this mode of usage, it doesn't carry a

Listing G.4.1:2: A snippet of syslog messages, converted to a .csv file by `numLogfile2Csv.py`

```
# [,start,start2,end,end2,size,value,value2
134868.420862,0,0,0,0,11,0,0
134868.421314,0,0,1,1,11,1,1
...
134868.425030,9,4,8,3,11,19,4
134868.425216,10,0,9,4,11,20,0
134868.425403,0,0,10,0,11,21,1
...
```

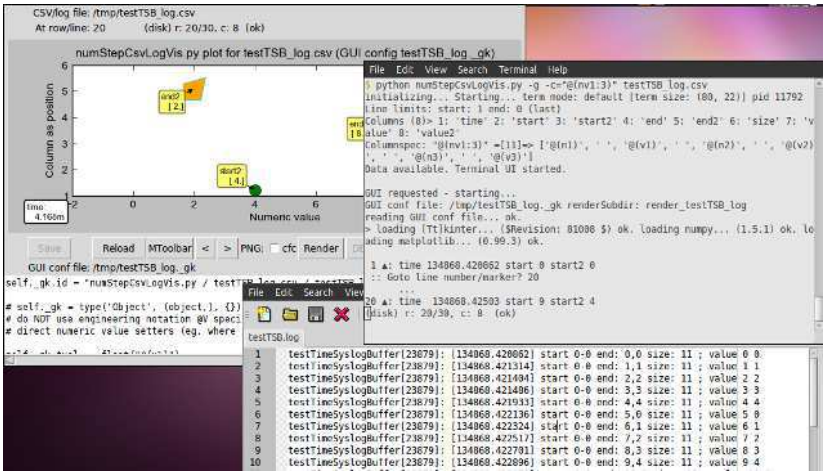


Fig. G.10: Screenshot of `numStepCsvLogVis.py` running - a typical log shown in a text editor (bottom); the terminal part of the application stepping through the CSV representation of the log (upper right); the `matplotlib` GUI part of the application (upper left)

dependency on a GUI toolkit. In the most basic form, it allows the user to "step" through the read .csv lines, by pressing the arrow keys on the keyboard: at each step, the current log line will be printed at the bottom of the terminal (with the rest of the lines scrolled up, as is usual for a terminal). However, using a CSV offers the "a priori" knowledge of the number of columns, and their names, in the data file – and as such, allowed us to implement a small "column specification" formatting language. Such a "columnspec" can be specified on the command line, as on listing G.4.1:3.

For instance, the columnspec "`@(nv:2)`" will expand with the name and value of all columns up to 2 (here, "`time`" and "`start`"), separated by space; and "`@(nv5-[self.currow-1])`" will expand to the name, and the current value minus the previous row's value, of column 5 (here, "`end2`"); all other parts of the columnspec string are used verbatim in the formatted output. Notice on listing G.4.1:3, that the software tries to keep track of the character width of all the numeric values, and allocate space for them accordingly; stepping through the entire CSV log file once, would inform the software about the max-

Listing G.4.1:3: A snippet of output during `numStepCsvLogVis.py` stepping, with a "column-spec" as a command line argument

```
$ python numStepCsvLogVis.py -c="@(nv:2) ; delta: @(nv5-[self.currow-1])" \
    testTSB_log.csv
...
20 ▲: time  134868.42503 start 9 ; delta: end2   1
21 ▲: time 134868.425216 start 10 ; delta: end2   1
22 ▲: time 134868.425403 start  0 ; delta: end2  -4
...
```

imum character width each column value requires, and thereafter the column formatting would be constant.

The application also supports a `curses` mode; a real-time "player" that can play back time-stretched timestamped data; an animation mode where a `Matplotlib` plot can be generated at each step, and exported as a sequence of bitmaps; and there is an example on how to sonify data, allowing for creation of videos with sound. All of this functionality was implemented, to reduce the cognitive strain during analytical browsing of arbitrary plain-text log files, specifically in the case of timestamped data – so that it would be easier to catch an error "in the act", so to speak. We have released `numStepCsvLogVis`, which runs under both Python 2.7 and 3, as open source in 2013; and more information and download links are available online via [1].

G.5 Developing a virtual, CD quality, ALSA driver

Having reached the conclusions in Sect. G.2, Sect. G.3 and Sect. G.4, we ported our AudioArduino driver to use the high-resolution timer API in the Linux kernel – again with the goal to change its settings from the original 8 bit/44.1 kHz/mono to CD-quality (16 bit/44.1 kHz/stereo). In doing this, we ran into expected problems, such as faulty buffer and period wrapping arithmetic, which eventually got resolved. However, that resolution wasn't complete: while problems with ALSA-only software (like the command-line applications `aplay` or `arecord`) were not detectable anymore, full-duplex operation in `audacity` kept on exhibiting problems, such as random drops, or insertions of snippets of silence in the capture stream. It should be noted that in our projects like [3a], one of the main goals with AudioArduino is to afford a potential student of digital audio a full-duplex operation experience in a high-level audio software (here primarily exemplified by `audacity`), initially through a duplex loopback test (which involves the Arduino device simply copying the playback data into the capture stream); thus, this kind of a problem represented a show-stopper for us.

We initially understood this behavior to be indicative of problems with our driver programming: a failure to take into account some unknown parameter,

that would become relevant first at CD-quality rates, exposing itself only due to the increased load implied by the use of `audacity` (along with its additional level of indirection into ALSA, due to its use of the PortAudio library). To confirm this, we felt that we would need to compare our driver against a sort of a benchmark, a driver of similar characteristics: one that uses the high-resolution timer API, and works at CD-quality rates. However, the hrtimer API is used in only a couple of drivers in the ALSA source files (among them the virtual `dummy` driver); and even if we had access to those particular soundcards, we may not have been able to use them on our development platform (as netbooks lack a traditional PCI slot, for instance). Thus, we found the idea, of a virtual driver – a version of the `dummy` driver, in the spirit of `minivosc` [2a], but working at CD-quality rates – used as a benchmark to compare against, attractive.

Note again that the original, virtual `dummy` driver doesn't perform any memory operations, which is why it wouldn't suffice as a benchmark to compare against; our version, provisionally called `dummy-mod`, just like `minivosc` writes (copies memory) to the capture stream (and thus `minivosc-mod` would have been a better name choice for it, if it wasn't for the fact that the two use different kernel timer APIs). Having fully expected that the virtual driver `dummy-mod`, would demonstrate a proper full-duplex behavior and work "out of the box" at CD-quality, imagine our surprise when also *this* driver exhibited problems, shown on Fig. G.11.

The situation on Fig. G.11 is notable, because it raises the question: *why* would a *virtual* driver operation cause a failure *at all* – even if the failure is in a component (PortAudio) which is not directly a part of the ALSA framework? Or, alternatively: why would a virtual driver cause an error, that drivers for actual hardware (like `hda-intel`) do not cause, on the same platform (and for the same stream quality)? In more detail, the top stereo track on Fig. G.11 is a sinusoid waveform generated from within `audacity`, but could be arbitrary audio data; it is simply used to activate the playback process along with the capture one, so the virtual driver performs in full-duplex mode; that playback data is otherwise ignored, and has no further effect in the full-duplex operation. The capture data – the bottom track – is generated by the virtual `dummy-mod` driver itself: upon each start of audio operation, the ALSA framework sets up "buffer" and "period" sizes in bytes for a driver, the "period" size corresponding to the amount of bytes to be transferred per call of the periodic timer function. Our virtual driver then writes a "big" sample pulse at each buffer boundary, and a "small" sample pulse at each period boundary; typically, `audacity` in full-duplex mode sets up two periods per buffer, and thus we expect alternating "big", then "small", pulses in the captured audio stream. This is why the emphasized part on Fig. G.11, showing a "big" pulse followed by two "small" ones, is a problem: it implies that a period's worth of bytes – a period right after a buffer boundary – has been altogether removed, or dropped. Even worse, Fig. G.11 shows this drop happened before 3 seconds expired, since the start of the full-duplex operation.

Confirming that this is a failure occurring in PortAudio, was again a costly

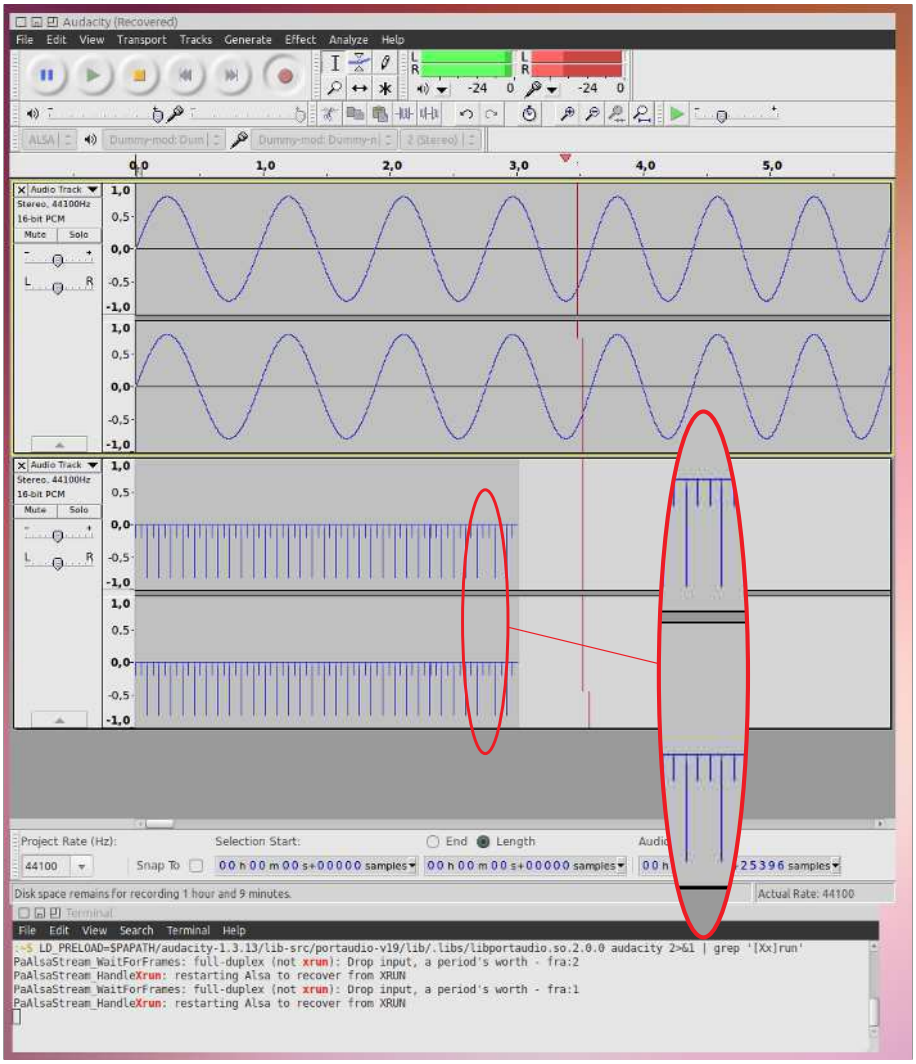


Fig. G.11: Screenshot of a drop with a virtual soundcard driver (`dummy-mod`, our modification of `dummy`) triggered in PortAudio, during CD-quality full-duplex operation of Audacity (the vertical breakage of the Audacity cursor is an artifact of the screenshot process). The top audio track is used to drive the playback operation; the bottom track is the captured data, with the drop emphasized; the terminal below shows related messages from PortAudio.

endeavor for us; one needs to start suspecting PortAudio to begin with, before rebuilding the library in debug mode, to enable printout of additional debug messages. Unfortunately, the part of the PortAudio code that triggered this behavior originally did not contain any debug messages whatsoever, and we eventually found it by inserting additional debug printout messages - and observing the correlation with the drops generated by the real-time full-duplex operation of `audacity` (instructions and code for reconstructing this behavior, as well as a patch file showing our added debug messages to the PortAudio library in this case, are available via [1]).

There are two conditions, buffer overrun and buffer underrun, that typically cause errors in data streaming; PortAudio code may refer to either as an "xrun". The PortAudio code, triggered during a drop as on Fig. G.11, states that the condition for entry in it is "not an xrun"; however it still results with handling typical for an xrun, which is dropping data altogether and restarting the stream. Thanks to the assistance of `alsa-devel`, `portaudio` and `audacity-devel` mailing lists, documented in [17], we eventually found that this is related to polling of file descriptors by PortAudio; and that one solution is to have the virtual driver's timer functions emulate the behavior of the `hda-intel` soundcard driver. However, before we focus on outlining this, let us first revisit the ALSA architecture in more detail in section G.5.1.

G.5.1 Yet another overview of an ALSA-based audio system

To appreciate the complexity that is incorporated in the ALSA system as a whole, consider Fig. G.12, which is a composite of several other diagram depictions.

Fig. G.12 shows three distinct diagrams, depicting the ALSA framework in different ways; note that in this figure, references to the older Open Sound System (OSS), found in the original source figures, have been removed. The diagrams are stacked along the z axis – however, all of them preserve the boundaries between the hardware, kernel- and user-space domains (along the y axis).

The top layer (from ref. [20]) depicts native ALSA user-space applications (or applications conforming to the ALSA API) utilizing the user-space ALSA library API, which as mentioned, is provided on a running Linux system by the `libasound.so` shared object file (not depicted on diagram). Besides the user-space applications, there may be libraries like PortAudio or PulseAudio which utilize the ALSA library API, and in turn provide a different API to user-space audio applications; and as such, add another level of indirection for user-space API calls to the ALSA library. The ALSA library API, in itself, may provide a "direct" access to soundcard hardware - or may allow usage of plugins (for audio conversion, routing, etc.) before the hardware is accessed. The user-space ALSA library then utilizes the ALSA kernel API in order to provide the actual audio operation functionality. The ALSA kernel API, in fact, provides a framework for four different types of audio functionality: PCM, MIDI, Control

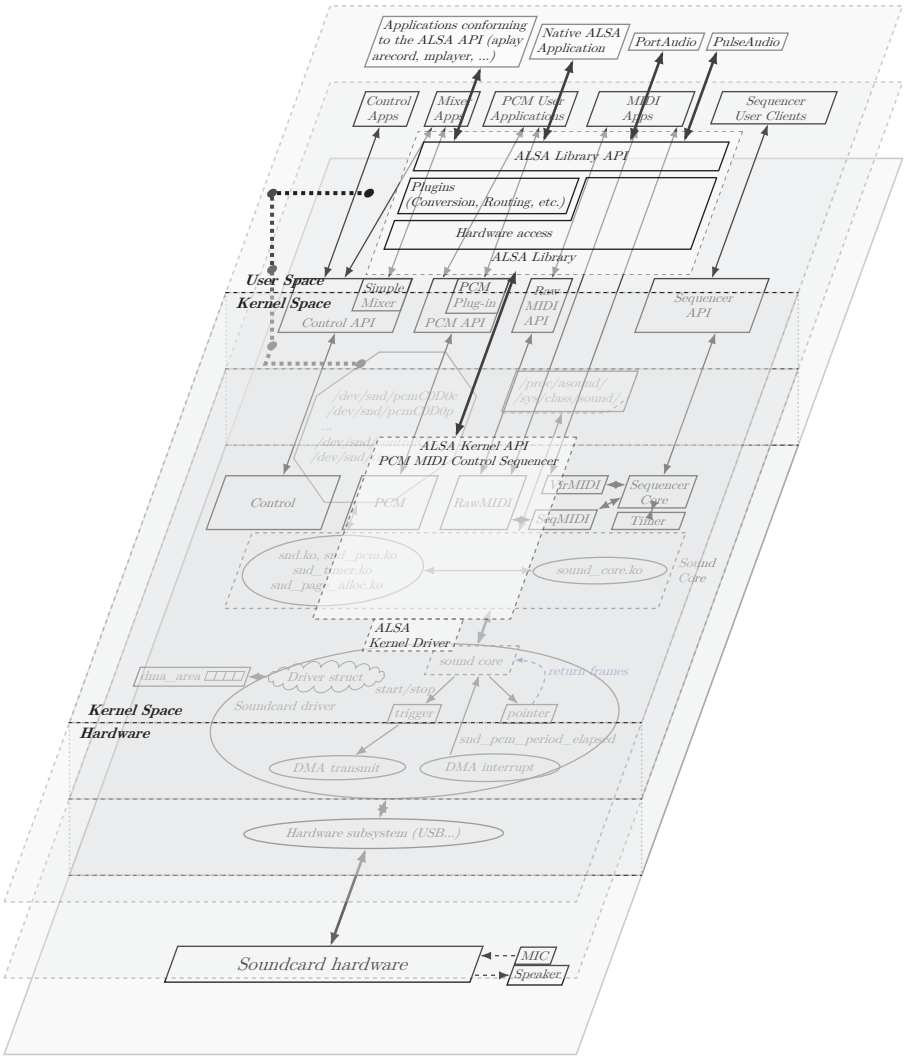


Fig. G.12: A composite of several ALSA diagrams, stacked along the z-axis. Bottom layer: ALSA subsystem from ref. [7] and ALSA driver from ref. [18]; middle layer: ALSA user- and kernel-space architecture from ref. [19]; top layer: ALSA basic structure/framework from ref. [20].

and Sequencer; finally, this framework is completed by the ALSA kernel driver, which implements the part of the functionality as specific to the particular soundcard hardware.

Let us mention that in all our ALSA related work so far, we have focused *solely* on the PCM functionality, as that is the part of the framework dealing with actual digital audio data streaming. Note that PCM typically stands for "pulse-code modulation", which as a term, could refer to a specific binary encoding (e.g. ITU G.711 A-law or μ-law [21]) as used in traditional telephony (like in "four-wire" telecom PCM equipment, as described in e.g. [22]) – but may also refer generally to "*the basic concepts of transmitting a sequence of symbols, i.e., pulses, to represent information* [23]", applicable even to telegraphy. In ALSA (similar to other digital audio contexts, e.g. as in [24]), it is meant to refer to the fact that this portion of the framework deals with digital representation of analog audio samples. However, this digital sample representation in the PC memory depends on the driver audio settings (e.g. interleaved vs. non-interleaved), and is not necessarily the same as the telephony encoding the term PCM may specifically refer to.

The diagram in the middle layer of Fig. G.12 (from ref. [19]) shows a slightly different view of the ALSA architecture. Here the user-space applications are categorized as Control Apps, Mixer Apps, PCM User Applications, MIDI Apps and Sequencer User Clients. These applications would interact with the user-space API, which here is categorized in Control API (containing a SimpleMixer interface), PCM API (containing a PCM Plug-In), Raw MIDI API and a Sequencer API - we can consider these to be different interfaces of the ALSA user-space library API. These interfaces, in turn, hook into the kernel space APIs, here categorized as Control, PCM, RawMIDI, SeqMIDI, VirtMIDI, Sequencer Core, and Timer. The diagram in the bottom layer of Fig. G.12 shows the ALSA subsystem perspective from ref. [7]. Here the kernel-space "Sound Core", composed of the kernel modules `sound_core.ko`, as well as `snd.ko`, `snd_pcm.ko`, `snd_timer.ko` and `snd_page_alloc.ko`, exports generic ALSA information in the `/proc/asound/` and `/sys/class/sound/` "pseudo filesystem" subdirectories of the root filesystem of the machine. Finally, the diagram on the bottom layer of Fig. G.12 features an overview of the organization of an ALSA kernel driver (from ref. [18]), which the sound core uses to hook into a particular hardware subsystem that will eventually interact with the soundcard hardware - we will focus more on this in a bit.

Following the UNIX philosophy that "everything is a file" [25], the sound core exports files that represent the substreams of a given soundcard hardware (which is present, and for which a driver is loaded) - "pseudo" files like `/dev/snd/pcmC0D0c` or `/dev/snd/pcmC0D0p`, which represent the capture and playback streams (respectively) of sound card 0, device 0 (see [26], or /Documentation/sound/alsa/ALSA-Configuration.txt in kernel source) as seen by ALSA on that particular system. Note that the kernel exports these files, only if ALSA is present and running on the system; via [1] we provide a script, `load-alsa-debug-modules.sh`, which shows how the ALSA kernel modules

can be blacklisted, so that ALSA is not active after the OS boots; and also shows how either "vanilla" ALSA modules - or debug ALSA modules (from a different, custom, location) - can be loaded into a live system. As it can be seen in the `load-alsa-debug-modules.sh` script, for our 2.6.38 development OS and platform, there are a total of 15 kernel modules that constitute the ALSA engine in kernel space; three of which deal with the onboard Intel HDA soundcard present on these PCs – however, on the 2.6.32 platform, there are differences (e.g. there are 18 kernel modules, including some OSS compatibility modules which are not present on the newer kernel). Note that if ALSA modules are loaded upon boot, it may not be possible to remove them - since it is not possible to remove a kernel module from Linux, which has a refcount (reference count of registered users of the module) different from zero. These versions of Linux can correctly resolve refcounts due to kernel-space module dependency, and for some user-space calls (like module loading and unloading) - but unfortunately not for *all* user-space calls to the kernel module: some user-space calls will cause the refcount of a module to increase, without a possibility to decrease it afterwards. Therefore, having the ALSA kernel modules loaded at boot time, almost guarantees that some user from the operating system (e.g. PulseAudio) will claim the drivers at startup, making it impossible to remove them in the same running OS session - which is what forces the blacklisting step. Additionally, note that different Linux distributions may choose to build ALSA monolithically "in-kernel", as opposed to building it as separate loadable kernel modules - in which case it should be impossible to remove ALSA (as there are no separate kernel modules anymore, whose loading at boot time can be blacklisted).

It is important to note that the ALSA library API essentially allows access by user-space applications to soundcard hardware *through* the exported PCM files; that is the reason we have included the dashed line on Fig. G.12, which connects the ALSA library box on the top layer diagram, with the exported files hexagon on the bottom layer diagram. This allows us to equivalate the audio operations to file operations: soundcard playback would be equivalent to writing data to the `/dev/snd/pcmC*D*p` files, while soundcard capture would be equivalent to reading data from the `/dev/snd/pcmC*D*c` files (or rather, the corresponding file descriptors). However, while [26] (from 1999) may imply that, say, `/dev/snd/pcmC0D0p` is directly writeable – we would have to peruse the `alsa-devel` mailing list archives, to realize that `/dev/snd/pcmC*D*p` files have not been directly writeable since at least 2003, because "*the native devices need a special initialization using ioctl* [27]". Thus, in order to interact with these "pseudo" files, we would have to use the ALSA Library (or `alsa-lib`) user-space `C` API (although, note that writing to "pseudo" files such as `/dev/audio` may still be possible through OSS emulation).

The main resource describing the different methods of interacting through these files from the ALSA library API is [28]; to begin with, it notes that there are three UNIX environment transfer methods (ultimately, in respect to file descriptors): *standard I/O transfers*, exemplified by functions like `read` and

`write`, which can have blocking or non-blocking behavior; *event waiting routines*, exemplified by functions like `select` and `poll`, which allow reception of notifications from the underlying device; and *asynchronous notification* which allows data transfers to occur in a handler of the `SIGIO` signal, and is related to the UNIX `signal` function [29]. This relates to the following types of ALSA transfers, exemplified through user-space API functions:

- *Standard Read/Write transfer* – `snd_pcm_writei()` and `snd_pcm_readi()` for interleaved, and `snd_pcm_writen()` and `snd_pcm_readn()` for non-interleaved access
- *Direct Read/Write transfer* (via mmap'ed areas) – involves calls to `snd_pcm_mmap_begin()` at start, `snd_pcm_mmap_readi()` and `snd_pcm_mmap_writei()` for interleaved (or `snd_pcm_mmap_readn()` and `snd_pcm_mmap_writen()` for non-interleaved) access, and `snd_pcm_mmap_commit()` to acknowledge end of transfer
- *Asynchronous mode* – in which `snd_async_add_pcm_handler()` can be used to define a handler, where the above transfer functions would run; then, the function `snd_pcm_poll_descriptors()` can be used to obtain file descriptors for event waiting routines.

These transfers could be used from single-threaded or multi-threaded user application contexts, and could run in blocking or non-blocking mode. Before any of this can be used, the programmer needs to open the corresponding soundcard device, by using a string of the form `"hw:0,0"` (which contains card and device number) in the call to `snd_pcm_open()`, and then set its hardware and software parameters; transparently to the programmer, this will set up the utilization of the underlying PCM pseudo files of the device. The resource [28] also refers to basic `C` code examples (in the `alsa-lib` sources) which demonstrate the usage of this API; earlier, there were also some ALSA sample programs in [30], which demonstrated up to six combinations of the above methods per example (unfortunately, at the time of this writing, the resource [30] seems not to be available on the Internet anymore).

Utilizing these functions in user-space (and the respective PCM pseudo files), results with eventual interaction with the ALSA engine in kernel space – and in particular, the respective soundcard driver module. The driver consists of several callback functions, which run at pre-defined times (e.g. when the driver is loaded or unloaded; when the hardware device is attached to or removed from the given bus; upon start and stop of audio operation issued from user-space software, etc; we have included more details on this in [2a, 3a]). On the bottom layer of Fig. G.12, two of these callbacks: the `.trigger` and the `.pointer`, are shown. The `pointer` callback's job is to return the amount of audio samples (expressed in units of ALSA frames), that have been played back or recorded up to that point in time, to the sound core – and it eventually allows the rest of the audio engine to do some error checking (e.g., if an xrun condition

has occurred). Part of this error checking is triggered when the driver calls the `snd_pcm_period_elapsed()` function (of the ALSA kernel API) – which, in this framework, the driver is obliged to do periodically. The different callback functions in the driver can exchange data through a `C` structure which can be provisionally called a "driver struct": it might contain both primitives (integers, strings), and pointers to memory or other structures, and it is created at the discretion of the driver programmer; as such, it can become quite complex, which is why in [2a, 3a] we have attempted to visualize the respective driver structs as structure maps. One of the most important pointers the driver struct stores, is the pointer to the portion of memory through which the driver exchanges data with the soundcard hardware, known as `dma_area`: each capture or playback substream has its own `dma_area`, and ALSA copies data from userspace to this memory area for the soundcard playback, and conversely it copies data from this memory area to userspace in the case of soundcard capture. In [3a], data from the playback `dma_area` is used as input for the `ftdi_write` function, whereby the playback data copied and sent to the USB chip; while data received in the USB callbacks from the chip is copied to the capture `dma_area` – in this way, the driver utilizes the USB kernel subsystem to, finally, address the actual soundcard hardware.

Having this in mind, let's take a closer look at the concept, whose demonstration has been a key goal in our driver development – that of full-duplex audio streaming.

G.5.2 Frames, periods, and the meaning of full-duplex

Before we continue, let's review how digital audio (PCM) data storage is handled in ALSA. By setting an audio sampling rate $f_A = 44.1\,\text{kHz}$, we specify that 44100 samples are to be transferred each second - for *each* independent audio channel. A sample can also be stored with different sizes (which relates to the sampling resolution); we can choose to store each sample, say, as an 8-bit unsigned integer (like in [2a, 3a]). However, for a CD quality reproduction, samples are stored as 16-bit, little endian, signed integers - and as such, each sample takes up two bytes. Furthermore, multiple channel data is typically *interleaved*, meaning that for a given sample time, a sample from each channel is taken and grouped together in a sequence, known as an ALSA *frame*. In other words, an ALSA frame is an "*equivalent of one sample being played, irrespective of the number of channels or the number of bits* [31]"; or a "*set of samples, one per channel, at a particular instant in time* [32]" (see also [28], [33]). In terms of data transfer, this allows us to understand the audio rate sampling rate of 44.1 kHz as a requirement for transfer of 44100 frames, regardless of the number of channels and sampling resolution involved. This kind of organization then determines the layout of audio data storage in memory, shown on Fig. G.13.

An ALSA *substream* is a software structure that roughly corresponds to a hardware connector on the soundcard: a driver for a soundcard with one stereo playback and one stereo recording minijack plug, would declare one stereo play-

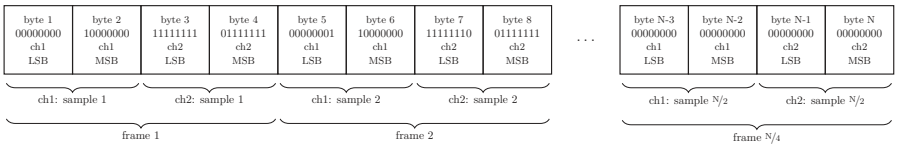


Fig. G.13: Byte-level layout of a 16-bit, little-endian, interleaved, stereo (two channel) digital audio (PCM) data in computer memory (e.g. a `dma_area` of an ALSA substream). Larger units like samples and ALSA frames are indicated, labeling uses 1-based indexing.

back substream and one stereo capture substream. Each substream refers to its own `dma_area`, and thus Fig. G.13 would illustrate data stored there, in case of stereo, 16-bit, interleaved setup - however, it is also a format used to store data in Microsoft `.wav` files. Noting that endianness is only relevant for numeric values that are larger than (and thus cannot fit in) a byte, a consequence of it is that if we read the first values in Fig. G.13 as 8-bit unsigned integers, we get $00000000_2 = 0_{10}$, $10000000_2 = 128_{10}$, $11111111_2 = 255_{10}$, $01111111_2 = 127_{10}$, etc. – however, if we read them as little-endian 16-bit signed integers, we get $1000000000000000_2 = -32768_{10}$, $0111111111111111_2 = 32767_{10}$, etc as sample values (see the script `print_sawramp.sh` in [1] for an example of generating and inspecting such a file; and Fig. G.24 for an example waveform plot). Since the signed integer values are dimensionless, as first approximation into their physical meaning we can take that the bounds of the 16-bit signed range are linearly mapped to the positive and negative peak voltages, that an eventual digital-to-analog converter (DAC) would reproduce. An ALSA driver would use the constant `SNDRV_PCM_FMTBIT_S16_LE` to declare itself capable of handling little-endian, 16-bit, signed integer audio samples; and `SNDRV_PCM_INFO_INTERLEAVED` to declare capability for interleaved audio data access - so for CD-quality settings, it holds that N frames, would correspond to $2N$ samples per channel, or $4N$ bytes in all (or: here one frame is equivalent to four bytes).

The audio data rate of 44100 frames per second has to be implemented through periodic transfers of data between the PC and the soundcard hardware, but it in itself doesn't specify at what actual period of time will data be exchanged. From a user-space perspective, the ALSA library API offers functions to set this explicitly - but it also offers functions like `snd_pcm_hw_params_set_period_time_near` and `snd_pcm_hw_params_set_period_size_near`, which would set up the value, nearest to the requested, that the soundcard is capable of. From a kernel-space perspective, the driver module declares the capabilities of the soundcard through the `period_bytes_min` and `period_bytes_max` properties; and in the `.prepare` callback (which runs at each start of an audio operation), the finally "decided" period size is reported to the driver through the `substream->runtime->period_size` property (given in unit of frames). Since the period size should implement the audio rate, it also determines the period time: if the audio rate is $f_A = 44100\,\mathrm{Hz}$ (or frames per second), and

the period size $\texttt{period\_size}_f$ is (say) 2048 frames, then the period time T_P (in seconds) can be calculated via:

$$T_P = \frac{\texttt{period\_size}_f}{f_A} \cdot 1\,\mathrm{s} \tag{G.5.1}$$

... resulting with (say) a period time of $T_P \approx 46\,\mathrm{ms}$. This is also how in a driver like AudioArduino [3a] we find the time period, at which the previously discussed periodic timer functions are set to run. Note that ALSA typically allocates the size of a `dma_area` to be an integer multiple of the period size; this is known as the buffer size, and is also reported in the `.prepare` callback of the driver, through the `substream->runtime->buffer_size` property (given again in unit of frames).

When a period time has expired, a period size amount of data should have been exchanged; the kernel driver is obliged to inform the rest of ALSA about this by calling `snd_pcm_period_elapsed()`. The kernel driver also returns the "current" or "latest" status of the data transfers by maintaining and returning a buffer position value from its `.pointer` callback function (expressed in frames) - which the rest of ALSA can call at any time. Additionally, ALSA maintains two more position pointers related to the same buffer, accessible also to user-space: `substream->runtime->status->hw_ptr` which generally follows the value returned by `.pointer`, and `substream->runtime->control->appl_ptr` which indicates the buffer position, that the user-space application has been able to handle up to that point in time. A setup of 4 periods per buffer and corresponding position pointers is shown on Fig. G.14.

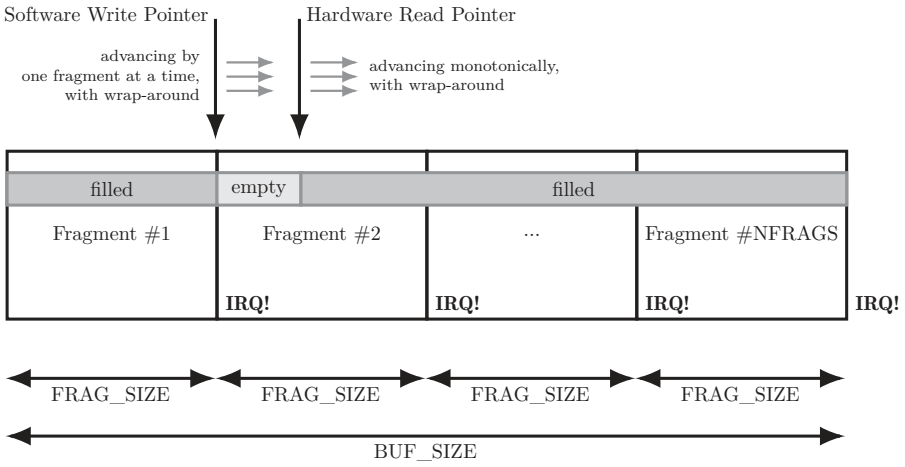


Fig. G.14: Schematic overview of the playback buffer in the traditional playback model, valid for ALSA (from ref. [34])

Figure G.14, and its source article [34], emphasize several important points

that may otherwise be easy to overlook in other ALSA documentation. First, note that Fig. G.14 uses the term *fragment* (also used in the earlier Open Sound System), to refer to what ALSA calls a period. Essentially, Fig. G.14 would represent the entirety of the playback `dma_area` buffer, where the data would be arranged like on Fig. G.13; `BUF_SIZE` would represent the entire allocated size of the buffer, while `FRAG_SIZE` would represent the period size. An important point is that the audio playback process is conducted through the same memory buffer, but in two distinct stages: a user-space application software would have to *write* audio data in this buffer (audio data which could have been read from hard disk, or generated on the fly), while the soundcard, with some latency, would have to *read* audio data from this buffer in order to reproduce the audio samples. Thus, there is a Software Write Pointer (the `appl_ptr`), which keeps track of how much data has the user-space application written – and a Hardware Read Pointer (the `hw_ptr`), which keeps track of how many samples of that data have been played back by the card already. This implies that the software write part has to occur first, and thus the software write pointer should in principle be ahead of the hardware read pointer; thus, on Fig. G.14, the software write pointer being behind the hardware read one, implies that the software write pointer has already "wrapped" - given that the buffer is considered to be a circular or ring buffer. The area between the two pointers doesn't have to be empty (it might contain "old" data), however, in any case it doesn't contain up-to-data for usage; we could consider it as a region that is allowed to be overwritten by the software write process. In this context, an underrun would happen if the application is takes too much time to perform the software write parts of the process, thus allowing the hardware read pointer to "catch up" with the software write pointer (which would mean that there is no more valid data for the soundcard to reproduce).

We can conceptualize the capture process similarly, except some meanings change: `hw_ptr` would be a hardware write pointer, since the card has to write the recorded data in the `dma_area`; and `appl_ptr` would be a software read pointer, since the application has to read the `dma_area` before it ultimately saves its contents to disk. Another point that Fig. G.14 emphasizes, is that "*traditionally on most operating systems audio is scheduled via sound card interrupts (IRQs)* [34]". On the other hand, in [2a, 3a], we use timer functions to schedule audio; this is similar to the traditional model in the sense that kernel timers run as software interrupts (softirqs) - however, there are more fundamental differences, which are discussed in more detail in the next section. Note that if a user-space application uses an access method which involves "sleeping" on a file descriptor, the soundcard interrupt would also serve to "wake up" that part of the process - thus becoming a signal to activate, say, the "software write" part of the playback process. Also, determining the optimum buffer and period/fragment size (and thus time) is not straight-forward: shorter periods increase CPU utilization and thus power usage (which becomes relevant in mobile use: note that [34] is a proposal that, among other things, suggests use of longer periods largely motivated by power-saving) – however, longer periods

also increase latency.

At this point, let us revisit the concept of full-duplex operation. Telecommunication textbooks usually provide a rather basic definition, such as data being "*transmitted in two directions at the same time* [35]". Actual implementation can be in different ways: e.g. in RS-232 asynchronous serial, there are two dedicated wires, RX and TX (Fig. G.6), which are one-way (or simplex) in nature – and they allow the transmission of data, both from the PC host to the device, and from the device to the host, at "the same time". Starting from that definition, full-duplex in terms of ALSA would simply mean the ability to run audio playback and capture processes from a user-space application at the same time. Thus in [5a], we can demonstrate a full-duplex operation from the perspective of the PC - even if the USB interface chip used there, the FTDI FT245, has a bidirectional parallel bus which has to be contended, and so per definition, can only work in half-duplex mode (for more, see sect. G.6.1); but that success, as noted previously, is due to the relatively low audio data transfer rate. At CD-quality, we have to remember that full-duplex would involve two buffers (one for capture, and one for playback) with corresponding pointers as on Fig. G.14, and two sets of IRQs/timer functions. Ideally, the playback and capture hardware buffer pointers should be in sync, since in principle they indicate the absolute passage of time – however, having noted kernel preemption and the jitter it introduces, we should be aware that such synchronization can be implemented only approximately; especially since it is quite possible that, at times, the OS may fail to service an interrupt request (or a timer function) fully.

Eventually, the meaning of full-duplex with ALSA is slightly stricter than the basic definition: it also implies usage of specific API functions in order to set up streams for full-duplex. In an ALSA driver kernel module, one should use `snd_pcm_group_for_each_entry()` with `snd_pcm_trigger_done()` in the `.trigger` callback function, so this callback (as well as `.start` and `.stop`) runs only once for both streams (otherwise the playback and capture streams would have `.trigger`, and other callbacks, called separately twice). In user-space, one should use the ALSA library API function `snd_pcm_link` to link the playback and capture substreams; its use may propagate to other libraries as well (e.g. PortAudio uses it in its `PaAlsaStream_Configure()` function). This approach would minimize the discrepancies between the playback and capture hardware pointers, however it cannot eliminate them completely: on a single CPU system, the CPU would still have to execute IRQ handlers in series, even if the IRQ signals may have arrived at the exact same time - and this alone might introduce a small error (e.g. the playback and capture interrupt at 2048 frames; if capture gets serviced first, the playback has to wait; when playback gets serviced, maybe the soundcard has already advanced to frame 2049, but the callback may still refer to the value that originally caused the interrupt, that is, 2048). Finally, note that there may be subtleties regarding linked streams, depending on the use-case for the full-duplex operation (which is further discussed in subsection G.5.4, in context of the `latency.c` program

overview):

- "Monitoring" - if we want to listen to (play back) an audio signal that is just being recorded (captured)
- "Studio overdub" - if we want to "overdub", that is, record (capture) an audio signal over/in sync with already playing audio track

Interestingly, in our previous ALSA driver work [2a, 3a, 5a], we have never explicitly used the `snd_pcm_group_for_each_entry` full-duplex setup function, as the low data rates (max 44100 B/s for 8-bit mono streams) allowed those drivers to perform reliably even without it. Otherwise, that setup approach can be rather easy to overlook in the documentation; and we first became aware of it in the course of performing the driver comparison, described in the next subsections.

G.5.3 ALSA, DMA and timers: comparing HDA intel and dummy drivers

Up to this point, we have considered the architecture of ALSA, without mentioning one of the most important aspects of it: as hinted by default names such as `dma_area`, ALSA is primarily intended to address cards that exploit the *direct memory access* (DMA) mechanism to interface with a PC. To begin with, we should be aware that in modern commodity PCs, the CPU(s) communicate with random access memory (RAM) through a Northbridge chipset [36]. Part of such a mechanism, including the DMA controller, is shown on Figure G.15.

The DMA controller is a standalone integrated circuit (IC) chip; there are different manufacturers and models, but most common in introductory literature are Intel 8237, 8537 or 82357. The details of the interaction mechanism can be somewhat complicated, but a basic introduction can be found in [38], [39], [37]. To begin with, Fig. G.15 is modified so it fits the ALSA soundcard context: the peripheral device could be something else (like a floppy or hard disk controller), however, we have specifically decided to consider it as a soundcard here. We would expect the peripheral device to have its own crystal oscillator (XO) for timing its own, "card" events: e.g. in AudioArduino [3a], we derive a clock close to 44.1 kHz from the 16 MHz XO on the Arduino, which we use to reproduce or capture individual samples - these we would consider to be the "card" events (events occurring under card clock time).

On the other hand, Fig. G.15 notes that the PC also has its own XO as a master clock source, which controls the execution of events in kernel and user space. Since we have two different physical clocks in the form of crystal oscillators, we cannot expect them to run at the same rate (note an Arduino's 16 MHz oscillator frequency vs. a PC's 1.6 GHz), nor to remain synchronized: even if they were started at the same time, they will drift; in other words, the soundcard and the PC represent two different clock domains. Additionally, we would implicitly observe these two clocks from the perspective of our "own",

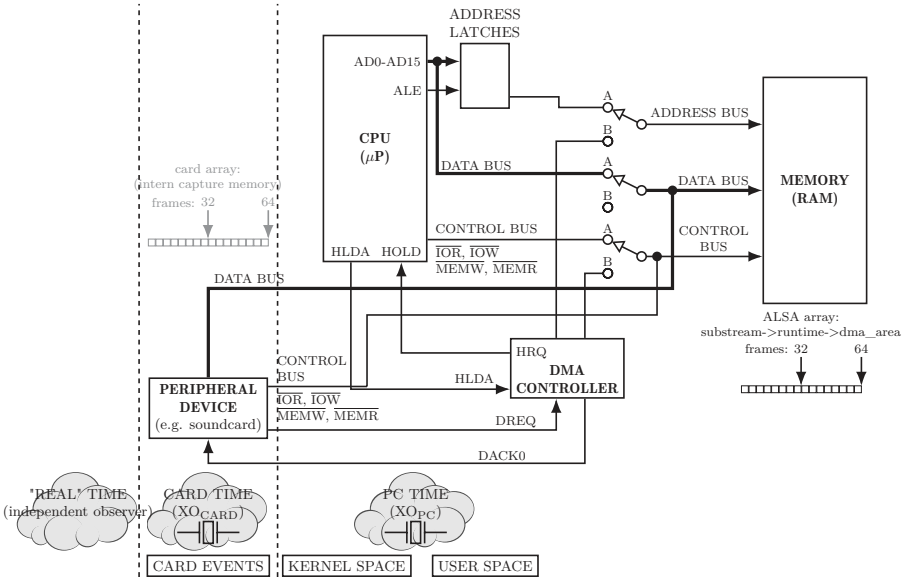


Fig. G.15: Block diagram of a DMA controller in a microprocessor system (from ref. [37])

independent clock, symbolizing the "real" flow of time - thus, in principle, we deal with three clock domains when observing a system like this.

Let us now assume that the soundcard on Fig. G.15 is performing a capture operation: analog-to-digital conversion, followed by storage of samples into internal capture memory of the card (here, as an example, 64 CD-quality ALSA frames in size). At the same time, the switches as shown on Fig. G.15 (in position A) allow direct communication of the CPU with memory through its address, data and control buses. Let's assume that once the soundcard fills a certain amount of its local capture memory (say, a period size of 32 frames), it will want to transfer this data to main memory. In this case, the card would assert the DREQ signal; as a result, the DMA controller would eventually ask the CPU to relinquish control of the bus, by asserting the HRQ signal. Once the CPU has set its bus pins to high-impedance (thus electrically isolating the bus lanes from itself), it confirms back to the DMA controller by asserting the HLDA signal. At this point, the DMA controller changes the switch positions (to position B), and sets up the correct control and address signals, so the card can output its data on the data bus while it is being correctly routed to RAM memory. The DMA controller then asserts the DACK0 signal, letting the card know to proceed with the transfer. Meanwhile, the CPU can process other commands, as long as they do not require information from RAM at that time. When the DMA transfer is done, the DMA controller starts by deasserting the HOLD (HRQ) signal, and eventually transfers the control of the bus back to

the CPU again.

Without this kind of a mechanism, the CPU would have to go through the entire instruction (or fetch-decode-execute) cycle [38], [40] for every single datum transferred; thus DMA allows for improved transfer efficiency while relieving the processor of wasting cycles. A consequence of this, is that the CPU is not "aware" of the data transferred via DMA: while certain DMA operations might be visible in a kernel log (like on Fig. G.2), it would be in principle impossible to, say, print out the data transferred *during* a DMA operation in such a log: one can only access (and print out) contents of RAM, after the DMA operation is complete. This, on the other hand, imposes interrupts as the most straightforward way for the card to notify the CPU of the progress of the transfer: since the CPU doesn't "know" the status of, say, the capture operation, the card can interrupt the CPU as soon as a period size of bytes has been transferred, which will act as a signal for the CPU to read the latest value of the buffer pointer, and possibly trigger further ALSA mechanisms accordingly. It is important to emphasize that in this case, for a capture direction, the value of the buffer pointer at the moment of interrupt handling would represent bytes that *have already been* transferred via DMA, and are present in the capture's `dma_area` in RAM – and this is what makes the DMA mechanism fundamentally different from the timer function approaches we've used in our previous ALSA drivers.

We first became aware of this, thanks to the following comment by C. Ladisch:

> "*Your driver's .pointer callback must report the* ***actual*** *position at which the hardware has finished reading from the buffer. You* ***must*** *read some hardware register of your DMA controller for this. It is not possible to deduce this from the current time because the clocks do not run at the same speed, and any kind of buffering will introduce more errors. The dummy driver uses a timer because there is no actual hardware.* [17]"

Further on in the same discussion thread, it is also noted that the assumption, that we made so far in the example – that a card would have an internal capture memory buffer, and DMA would transfer bytes between it and main RAM memory – is historical; and does not hold any longer for modern cards, for which "*all data is immediately read from/written to main memory* [17]" (this is indicated on Fig. G.15 with a grayed out "intern capture memory" symbol in the soundcard domain). Taking all this into account, it becomes clear that we would be ***always*** *late* when using a kernel timer callback as a periodic ALSA function, as opposed to using an IRQ handler of a real soundcard-generated interrupt; the chief causes being:

- An IRQ handler runs with the maximum kernel priority (that of a hardware interrupt); a kernel timer runs with the next lower priority (that of a softirq), and as such is more prone to jitter due to kernel preemption

- For the capture direction, the periodic execution of the IRQ handler signals existence of a period size of new capture bytes *already present* in the substream's `dma_area` in RAM; with a kernel timer function, we can at best just *start* the process of copying data into the `dma_area` (meaning, it will take additional time for the bytes to be actually present there) - and with a virtual driver, this copying process will have to utilize the CPU as well
- Since running potentially long operations (like copying memory) in a interrupt context (to which the kernel timer function, as a softirq, belongs) is not recommended, in both our virtual and AudioArduino drivers a tasklet is used as a "bottom half" [8], [7], to schedule such operations for slightly later; this can be seen to introduce a further source of jitter due to kernel preemption

To summarize: usage of kernel timer functions is not, and cannot be, an accurate simulation of the periodic performance of hardware interrupts of a soundcard that utilizes DMA, in an ALSA soundcard driver. However, currently we cannot conceive of a software based approach, that could be seen as a more accurate simulation (and would work without changes to a vanilla kernel of the series we used for development).

Confirming whether this is a correct understanding, however, can be difficult. To begin with, there is no "generic" DMA soundcard hardware (as on Fig. G.15; interfaced, say, via PCI), that we could use as a model; furthermore, a detailed investigation would require hardware measurement/probing of the high-speed, parallel PCI bus, something we are not in a position to perform at this time. Therefore we opted for something simpler: since both of our development netbooks feature an integrated onboard HDA Intel soundcard, we used this as our model of a DMA soundcard; and as an experiment, we produced a set of programs and scripts, that capture kernel log data while certain ALSA operations are performed, and then filter that data and plot it. The user-space ALSA programs (`captmini.c` and `playmini.c`) set up a card for CD-quality operation, with buffer size of 64 frames and period size of 32 frames (which results with a period time of $\approx 726\,\mu$s), and then call a quick succession of two ALSA commands (`snd_pcm_readi` for capture, and `snd_pcm_writei` for playback, respectively), before terminating; `ftrace` kernel data (as on Fig. G.2) is captured during execution of the two commands. Running these programs against different drivers – in our case, first a virtual `snd-dummy.ko` based one, then the `snd-hda-intel.ko` one – allows collection of data, which allows for the drivers to be compared.

We have chosen the small period and buffer sizes primarily to minimize the duration of operation. Even if the expected time for the operation to complete in this experiment is on the order of a few milliseconds, massive amount of kernel log data is generated, which needs to be filtered in ordered to be plotted meaningfully. We have chosen to limit the data to ALSA kernel functions (those prefixed with `snd_pcm_`, as well as functions defined in the drivers themselves),

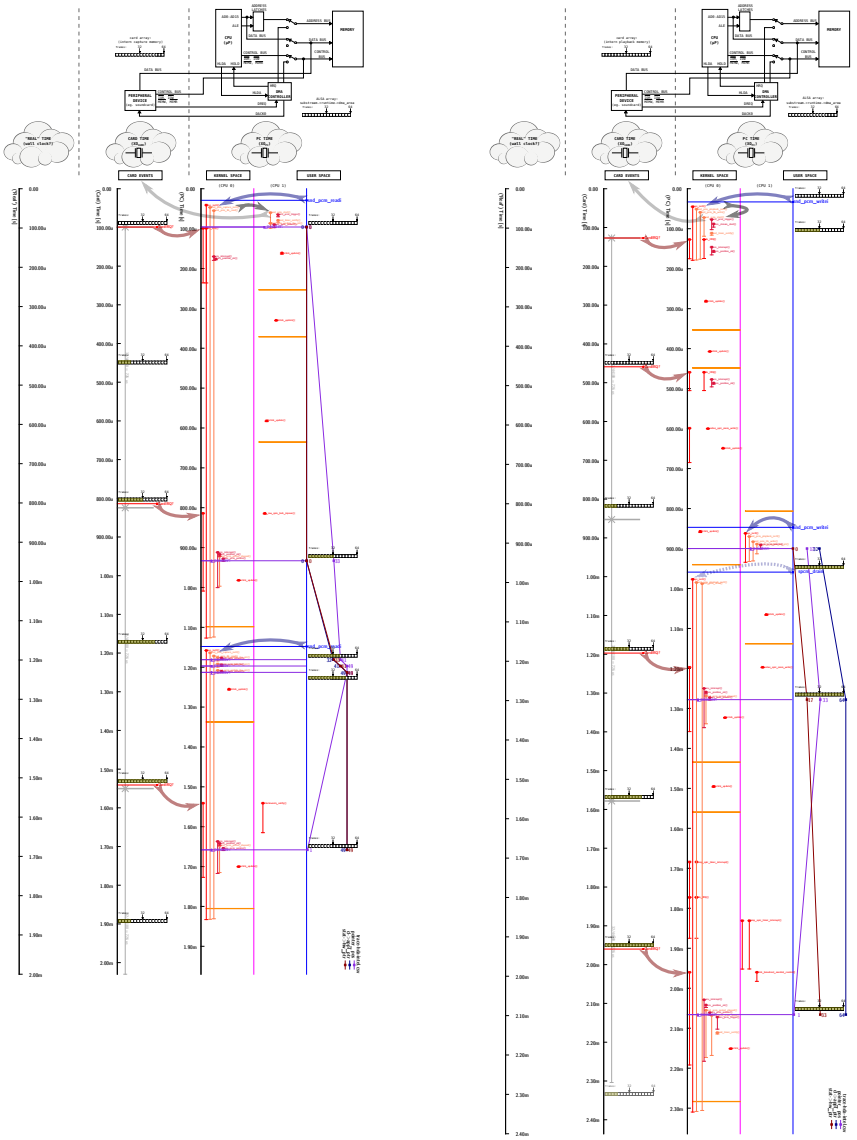


Fig. G.16: Behavior of the `hda-intel` driver: capture behavior with `snd_pcm_readi` (left) and playback behavior with `snd_pcm_writei` (right); blue arrows: userspace command, red: interrupt callback

ioctl handlers, IRQ handlers, task switching and some hrtimer related functions. Additionally, moments when the user-space functions run are plotted, and the drivers are modified so they print out the values of all substream buffer pointers when their `.pointer` function is called. Function names are arranged in per-CPU lanes with preserved call nesting indentation, and plotted on a vertical time line. Even with the filtering, it is very difficult to present the details in printed form – still, we submit Figure G.16 and Figure G.17, which plot the simplex behavior of the HDA Intel "real" soundcard driver, and our modified `dummy` virtual soundcard driver, respectively (on both, left side shows capture, right side shows playback); an electronic, PDF version of this article would allow sufficient zoom to observe details in these figures.

All plots on Figure G.16 and Figure G.17 feature the DMA controller and the time domains' symbols of Figure G.15 on top. Each plot presents a vertical time axis for each domain: as a visual reminder of the clock domain independence, the "real time" domain is taken to be the reference, the "card time" domain is shown to run faster than the reference, while the "PC time" domain is shown to run slower than the reference. The starting point is the start of logging taken as 0 s, aligned on all three axes, and time increases "downwards" (tick marks indicate 100 μs units). On the "PC time" domain, the first two lanes (from left) display filtered kernel functions from the `ftrace` log for each of the two CPUs; the third lane shows user space functions. The thick blue arrows from user to kernel space should indicate the kernel-space functions that run in response to the user-space function calls. On the "kernel time" axis, with thicker red vertical lines we indicate duration of any interrupt handler captured; while the orange rectangular backgrounds (often appearing as horizontal lines) indicate when the kernel switches user-space tasks. On the user space side, also a numeric indication (in frames) of the three ALSA buffer pointer variables is shown (the `.pointer` [violet], `hw_ptr` [red] and `appl_ptr` [blue] value), as well as a linear interpolation between their values at different points in time.

On Fig. G.16, the start of the IRQ handler (which contains the function which we consider periodic: `azx_interrupt()` for `hda-intel`, or `dummy_hrtimer_callback()` for `dummy`) is replotted on the "card time" axis 5 μs earlier, and considered as an interrupt request originating from the card ("card IRQ"). Thus, the card IRQ are shown to occur slightly "earlier" (in respect to "real time" axis) on the card, than their respective handlers running on the PC - which is what we'd otherwise expect to happen in reality too (the clock domain mismatch emphasizes this visually even more). In this sense, we try to extrapolate when the card interrupt requests have been raised on the card, based on the information we have on when the respective IRQ handlers ran on the PC. The thick red arrows on Fig. G.16 connect such a "card IRQ" in the card time domain with the respective IRQ handler in kernel space. Finally, we use a visual representation of a filled buffer: on the "card time" line, we've manually indicated the expected state of the assumed "card" buffer, while on the user space side, it indicates the value of the relevant buffer pointer variable.

In this way, Fig. G.16 reveals a fundamental asymmetry between the play-

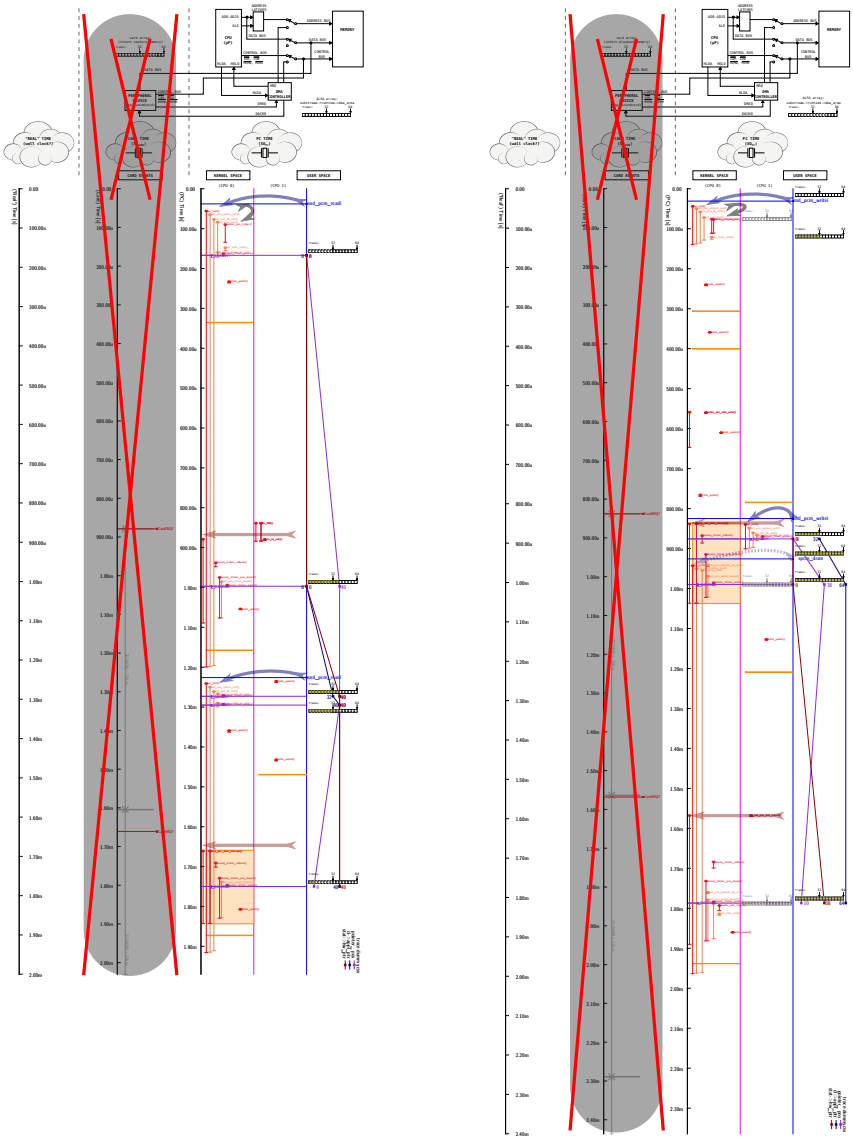


Fig. G.17: Behavior of the `dummy` driver: capture behavior with `snd_pcm_readi` (left) and playback behavior with `snd_pcm_writei` (right); blue arrows: userspace command, red: timer callback

back and capture ALSA operations: in a capture operation, the `.pointer` value (from the card) is the primary buffer pointer variable, `hw_ptr` follows the `.pointer` value, and `appl_ptr` follows the `hw_ptr` – while in a playback operation, the `appl_ptr` (set by user-space) is the primary buffer pointer variable, the `.pointer` value (from the card) follows `appl_ptr`, and `hw_ptr` follows the `.pointer` value. Essentially, the `.pointer`, since it originates from the card, is independent from the other two values, and its value informs the kernel about the status of DMA operations of the card: in the capture direction, it notifies how many frames have been captured by the card; in the playback directions, it notifies how many frames of those supplied by `appl_ptr` have already been played back. Then, `hw_ptr` always follows `.pointer`, but with a slight delay - it gets updated only at the call to the ALSA kernel function `snd_pcm_update_hw_ptr0`, and it gets updated to the *last* (previous) `.pointer` value registered by the kernel (at that point, the actual `.pointer` value is typically advanced from what `hw_ptr` is set to). The `appl_ptr` shows the user-space application status: how many frames the application has supplied for playback, or how many frames has it already copied from kernel-space for capture. Note that the `.pointer` value wraps at buffer size (in frames), as appropriate for the ring buffer nature of the `dma_area`; while `appl_ptr` and `hw_ptr` are cumulative (but are shown as if wrapping at buffer size on Fig. G.16 and Fig. G.17).

In other words, if we assume existence of card buffers, we could say the following on the operation asymmetry: in capture, the card buffer is filled first, the PC buffer (`dma_area`) follows it, and the "card IRQ" is a signal to the user-space application to start copying a period size of captured data; in playback, the PC buffer is set first by the user-space application, the card buffer follows it, and the "card IRQ" is a signal to the kernel informing that a period-sized amount of frames from those set by the user-space application have actually been played. Fig. G.16 reveals another asymmetry between playback and capture, which may be specific to the HDA Intel card and driver: in both cases, the first "card IRQ" is issued relatively quickly (on the order of 100 μs) after the first user-space command; however, for the capture case, we can consider it as a start of operation, and "card IRQ"s appear regularly at period time afterwards – while in the playback case, this first "card IRQ" seems like an acknowledgment, then another "card IRQ" runs after half a period time, which can be seen as the start of operation, and only afterwards do the "card IRQ"s appear regularly at period time. We cannot state with certainty the reason for this behavior of the HDA Intel card and driver - however, it does ultimately relate to our solution for a full-duplex, CD-quality virtual ALSA driver.

Another interesting point about this example is that `playmini.c`, for the small period and buffer sizes used, tended to generate lots of xruns when used with the `hda-intel` driver (note however that Fig. G.16 shows a successful run of the program, one where an xrun didn't occur). We made a small test using a script, `playdelay.sh`, which was based around inserting a given `nanosleep()` delay between the two `snd_pcmd_writei()` user-space ALSA calls, running

the program multiple times, and recording the number of xruns that have occurred. Thus we measured the response of delays from 100 μs to 1 ms in 10 μs increments, running `playmini.c` for 100 times in each increment; and realized that the optimal delay (the one for which a minimal number of xruns were recorded per 100 runs) between the two `snd_pcmd_writei()` calls is 310 μs, which is notably close to half the period time for this case (363 μs). To conserve space, we do not include the resulting plot in this article, but the script and the plot image can be found via [1] (or alternately, via [17]).

Fig. G.17 shows the same context as Fig. G.16, but for the operation of a virtual, `snd-dummy.ko` based, driver. Correspondingly, the card hardware and the "card time" axis are vividly crossed out on Fig. G.17, as no actual card hardware is used. The meaning of most symbols on Fig. G.17 is the same as on Fig. G.16, with the exception of the red thick arrow - it now cannot originate from the card hardware, so it is drawn pointing from the other side; and also it doesn't indicate a "card IRQ" anymore - it indicates where the timer function is running. In both capture and playback cases, at least a period time needs to expire after first user space command, for the first timer function to run; however, Fig. G.17 indicates that in the capture case, `snd_pcm_readi` may be blocking - that is, the first such user-space command may wait for an update from the periodic kernel function, before the next such user-space command proceeds; but in the playback case, it seems that both `snd_pcm_writei` commands can execute, before even the first instance of the periodic timer function has run - which implies queuing. Another thing visible on Fig. G.17 is that the period time isn't kept exactly, which shouldn't be surprising by now, considering the jittering behavior of high-resolution Linux kernel timers discussed previously (Fig. G.5).

While Fig. G.16 and Fig. G.17 can be said to illustrate some of the basic differences between a virtual (with periodic timer functions) and a hardware DMA (with periodic IRQs) ALSA driver, the level of understanding they bring about was still not enough to implement a virtual ALSA driver, that would work seamlessly at CD quality settings with Audacity as a front end. For that, we went through several other attempts at visualization of the kernel log data, while focusing on the methods the PortAudio uses to utilize ALSA - this is described further in the next section. Driver, program (`captmini.c` and `playmini.c`) and script code, as well as logs, used to create the plots on Fig. G.16 and Fig. G.17 is available via [1] (or alternately, via [17]).

Before we proceed, let us first return briefly to the PC soundcard hardware we used. To begin with, "HDA Intel" is a reference to Intel's "High Definition Audio", whose 225-page specification is publicly available [41]; this specification was codenamed "Azalia", which is likely the reason behind the `snd-hda-intel.ko` driver using `azx_` as a prefix for its driver functions. The specifications defines a "High Definition Audio Controller" device, which on one hand connects to the PCI bus, and on the other hand provides a high-speed serial bus called "High Definition Audio Link". Finally, there are devices that connect to the HDA Link bus, and otherwise contain the hardware A/D and/or D/A

converters, called "High Definition Audio Codec". There can be multiple such codecs attached to a HDA Link - and typically, their production is outsourced. Using the `alsa-info.sh` script (downloadable from the ALSA website), we have probed the soundcard hardware on our development netbooks, and realized they contain a Realtek ALC269 codec (NetColors) and a Realtek ALC662 rev1 codec (MSI Wind); the datasheet for ALC269 can be found on [42]. Thus, the model we have on Fig. G.15, where a single peripheral "card" would contain all circuitry (both the PCI interface and the A/D and D/A converters) is a simplification; in our actual case, the DMA soundcard hardware is split in two parts: a HDA controller by Intel, and a HDA codec by Realtek (with a serial bus in-between) - and correspondingly, there are several ALSA drivers loaded on a live Linux on our development platform: `snd-hda-intel.ko`, `snd-hda-codec.ko` and (ultimately) `snd-hda-codec-realtek.ko`. Still, inspecting the `snd-hda-intel.ko` was the right approach to obtain periodic interrupt information for Fig. G.16 – as this driver governs the soundcard system part (the HDA controller) which would be directly attached to the PCI bus.

G.5.4 Solving the virtual, full-duplex, CD-quality ALSA driver: Visualizing and animating ftrace kernel log files with gnuplot

The discussion in the previous subsection was a necessary, but not sufficient, prerequisite for implementing a virtual ALSA driver, which would operate under CD-quality settings, but without the drop problems in the PortAudio library (as illustrated on Fig. G.11). The user-space programs being traced in the previous subsection (`captmini.c` and `playmini.c`) implement capture and playback operations separately, and as such do not model full-duplex operation. Furthermore, they utilize only the simplest method of interacting with ALSA (via `snd_pcm_readi` and `snd_pcm_writei` ALSA API calls) in a single thread - while the PortAudio library polls ALSA file descriptors in a multi-threaded context. Because of this, we needed to identify additional user-space software for tracing inspection, which lead to additional approaches to visualizing data obtained from that.

We were interested first in finding a "minimal" full-duplex ALSA user-space program, in the same sense that `captmini.c` and `playmini.c` could be seen as "minimal" simplex user-space examples. While such software exists, its existence evaded us for a while: for one, it is only distributed in source form, in the `test` subdirectory of the `alsa-lib` source package; and then, its name is `latency.c`, making it easy to overlook in this context. Unfortunately, there was very little information about this program, beyond the source code itself, which mentions simply that it can be used to measure "latency between capture and playback". As such, understanding what the program does, and how, wasn't necessarily a straightforward process for us – and because of that, we wrote some of the understanding we gained as a sort of an initial documentation, and posted it as a page on the ALSA project Wiki [43].

The `latency.c` program is a single-threaded application, and ultimately reduces to using `snd_pcm_readi` and `snd_pcm_writei` again; however, it conveniently offers a way for choosing polled or non-polled mechanisms through command line arguments. On its Wiki page [43], we also included some quotes from relevant posts on the ALSA mailing lists; some of these further clarify the perspective that ALSA has on full-duplex operation. First, `latency.c` is intended to capture input data, and play it back as soon as possible; however, that can be implemented in two different ways, depending on whether `snd_pcm_link()` is used; which is shown on Figure G.18.

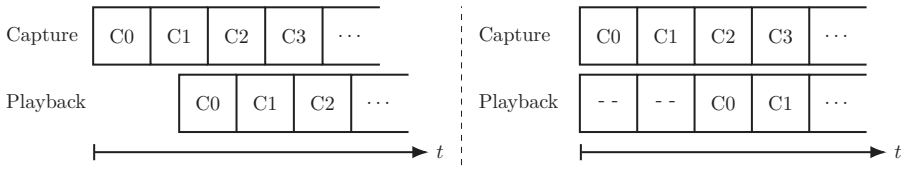


Fig. G.18: Difference between unlinked (left) and linked (right) full-duplex capture and playback ALSA streams for `latency.c` (from ref. [43])

Each of the square sections on Fig. G.18 represent a period size of audio data (their boundaries can be considered to be defined by the moments when the periodic functions [IRQ or timer functions] run). In case of unlinked streams, we have to wait for the first period of capture data (C0) to be available, before we can use that data for playback - however, since it will take additional time to start the playback stream from user-space, the delay between the start of capture and start of playback in this case will be greater than the period time. When the streams are linked, only one `snd_pcm_start()` command is used to start both capture and playback streams; in an idealized case, we can consider them "aligned" in time (as on Fig. G.18 right). Here, the first playback period must be silence (as at that time, the first capture period of data C0 is not yet available); and while we would like to start playing back C0 already in the second playback period, we cannot - because by the time user-space became "aware" that C0 is available, and copied to playback, the second playback period would have already started. This is the reason why the `latency.c` program starts with writing silence data twice to the playback buffer. Thus, ultimately, the latency that the `latency.c` program measures, corresponds to $2\cdot$ `period_size` plus whatever hardware latencies occur. Note that this behavior may be specific for this use case, which - in the terms we introduced at end of sect. G.5.2 - we could call a "monitoring" full-duplex use case (i.e. we want to play back data which is captured live as soon as possible); and may not necessarily apply to "studio overdub" full-duplex case (where we play back audio tracks, and we'd like to capture audio input, so it is - as much as possible - in sync with the playback data).

We used a modified version (with additional command line options, and other changes) called `latency-mod.c`, available via [1] (or alternately, via [17]),

for debug inspection. As it is ALSA-only application, analyzing it wouldn't track down the reason for the PortAudio full-duplex drop; however, we hoped to gain a better understanding of what happens on the kernel level during a polled full-duplex operation. Essentially, the analysis would consist of plotting data similar to Fig. G.16 and Fig. G.17; however here we would have to show both playback and capture data simultaneously - basically, as if the left and right plots on Fig. G.16 (or Fig. G.17) were to overlap. This makes it more difficult to read the plots and draw conclusions, and we attempted to address this by trying two animation approaches.

The first approach was to run a debug acquisition test multiple times, plot the acquired data as on Figs. G.16 and G.17, but in "landscape" mode (the plot is rotated so the time axes are horizontal), and then produce a GIF (Graphics Interchange Format) animation, where each animation frame is a plot of one test run; we used this approach both for `latency-mod.c` tests, and for those described in the previous subsection. With a low resolution GIF animation, details are lost – but, such an animation still allows for a visual understanding of the jitter present in the periodic functions: at first sight, one can deduce that the jitter of `hda-intel` driver's `azx_interrupt()` can be as large as the one of `dummy`'s `dummy_hrtimer_callback()`. More interestingly, it reveals that how often the driver's `.pointer` callback is called, can be dependent on the arrangement of both driver code, and user-space code.

In terms of driver code, let's recall that the three buffer pointer variables (`.pointer`, `hw_ptr` and `appl_ptr`), have different "primacy" depending on the operation direction (playback or capture); and do not change simultaneously, being updated by `snd_pcm_update_hw_ptr0` to a "previous" value. Because of this, the ALSA engine may typically call `.pointer` twice (or more) in quick succession - until the value of `hw_ptr` "settles" (that is, becomes equal to the `.pointer` value). This causes some interesting behavior: in the original `dummy` driver, the value returned for `.pointer` is based on measurement of expired time from start of operation; this means that there is a pretty good chance that by the time `.pointer` is called for the second time, enough time may have passed so that the calculated `.pointer` value has changed for 1 frame since last time. Thus, the `hw_ptr` cannot "settle" in the second call, and the ALSA engine continues calling `.pointer` in quick succession – "quick" here meaning mere 30 μs or so apart, which we may perceive as a useless utilization of the CPU (especially since this is a virtual driver, and there is no reason why, in principle, we couldn't return any value whatsoever for its pointer - including a value that would stop the quick succession). In the modified driver version we used, `dummy-mod`, we moved the calculation of the returned `.pointer` value in the timer callback; because of this, the `.pointer` value is updated less frequently - meaning that `hw_ptr` can usually "catch up" after only two calls to `snd_pcm_update_hw_ptr0`. In this sense, `dummy-mod`'s behavior resembles the operation of `hda-intel` more closely than the original `dummy` does. In terms of user-space code, we realized that if `latency-mod.c` is ran both without blocking and without polling, there will be a tight loop in user-space code

calling `snd_pcm_readi` - which will be visible in the animated GIF as many quick successions of `.pointer`, even for the `hda-intel` driver! Links to these GIF animations can be found via [1] (or alternately, via [17]).

The second approach to animation was to take debug log data acquired from a single run, and using a separate `gnuplot` script, render plot frames which represent a sort of a zoomed region into plots like Fig. G.16 (or Fig. G.17), centered around a particular time moment; one such frame is shown on Figure G.19. These frames are then composed as an MPEG animation, with the end result being a video of time-stretched vertical scrolling along the plot's time axes.

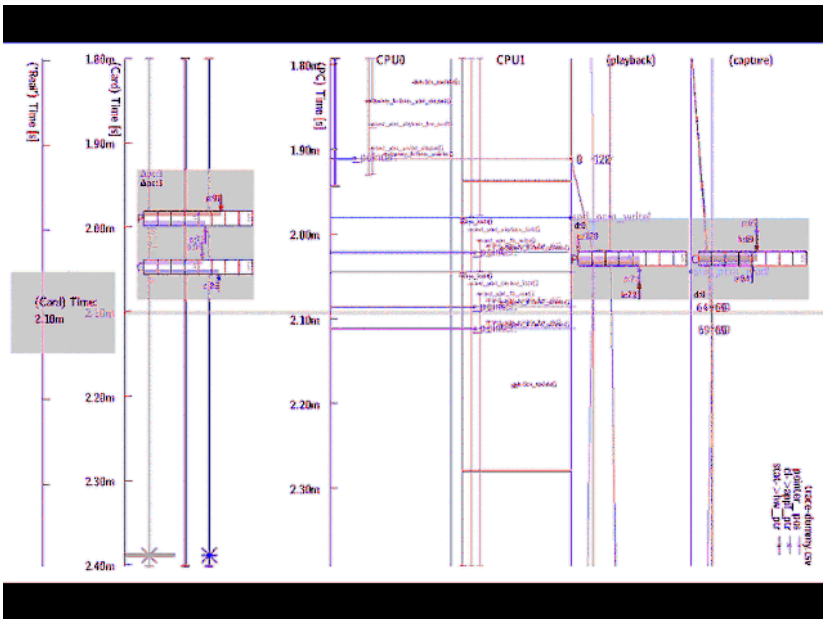


Fig. G.19: One frame of animation, of a kernel log of the full-duplex operation of the `dummy-mod` driver, used with the user-space `latency-mod.c` program.

The main benefit from an animation like on Fig. G.19 is that it allows one to gain a sense of the rhythm of the periodic kernel functions and the change of buffer pointers. The rhythmical sense could have been enhanced further by a sonification audio track (e.g. generated similarly to the approach in Sect. G.4.1), but we didn't pursue that further in this context. The animation is a time-expanded video (basically, a slow-motion video akin to high-frequency time-lapse videos) where $\approx 6\,\text{ms}$ of real-time debug capture is visualized as 26 seconds of animation. The plot is centered around the "current" "card time" moment (indicated by the captioned box and the horizontal gray line in the middle), and shows a range of $300\,\mu\text{s}$ before and after the "current" time. To the right, there are two lanes: one for playback, and one for capture; both

of them feature a (gray) box, with a set of squares representing the buffer (white background), and a rendering of the current value of the different buffer pointers: for playback, `appl_ptr` (dark blue) is on top as a primary variable, and the `.pointer` (violet) and `hw_ptr` (dark red) are on the bottom; for capture, `.pointer` (as primary) followed by `hw_ptr` is on top, while `appl_ptr` is on the bottom; all of the variables wrap at buffer size, and the value is indicated both as an arrow, and as a colored line inside the buffer symbol. To the left of the plot, there is again a similar box, with two buffer symbols: the top is related to playback, the bottom to capture. From the inner side of the buffer symbols, the respective `hw_ptr` variable is repeated (in violet); on their outer side, we show the assumed playback (red) and capture (blue) buffer pointer "on card", interpolated from the first known respective `hw_ptr` value at CD-quality rate. In an animated format, this allows for a more natural understanding of the buffer pointer mechanisms; unfortunately, it doesn't allow for more precise statements, other than `hw_ptr` generally being able to follow the assumed "card" pointer values (and the pointers following each other depending on stream direction as described previously), in both `hda-intel` and `dummy` driver cases.

Note that here we use period size of 128 frames and buffer size of 256 frames, since for smaller values, `latency-mod.c` very easily generates an xrun. In general, the smaller the period and buffer sizes, the more the likelihood for an xrun increases, and as such it also happens for the chosen 128/256 frames combination; however, we used this animation approach on test run acquisitions that ended successfully, in order to gain a model of how the proper operation is supposed to work. Having encountered xruns in both this and previous tests, made us briefly look into finding a way to visualize when an xrun is detected; unfortunately, as of now, we haven't found a simple and legible enough method. The best we can say currently, is that on the level of ALSA, the bulk of the calculations for an xrun detection seem to occur in the `snd_pcm_update_hw_ptr0()` ALSA kernel function, defined in `alsa-kernel/core/pcm_lib.c` file (in the `alsa-driver` package); and they may depend both on the period and buffer sizes, and on a `delta` defined either through new and old values of `hw_ptr` – or through an elapsed time (involving calculation via `jiffies`). Additionally, an xrun is detected in the `snd_pcm_update_state()` kernel function (in the same source file), where the detection relies on `avail` and `stop_threshold` variables. The `avail` variable, roughly, returns the difference between frames processed by user space, and frames processed by the soundcard; since the meaning of this differs depending on stream direction (playback or capture), there are different formulas and thus different functions to calculate this parameter: `snd_pcm_playback_avail()` and `snd_pcm_capture_avail()` (see file `alsa-kernel/include/pcm.h` in the `alsa-driver` package).

While the above animation approaches do manage to provide insights into the operation of ALSA drivers which are otherwise obscure, they didn't reveal the reason for the failure with PortAudio. Thus, we needed another user-space program, which would again be used to drive a full-duplex operation, but utilizing the PortAudio library - and in that sense, would emulate

the full-duplex audio operation of Audacity, without the GUI overhead. For this purpose, we used a program named `patest_duplex_wire.c`, which we developed from several PortAudio examples, most notably `patest_wire.c` and `patest_record.c` (found in the `test/` directory of the `portaudio-v19` source package). Note that in this program, as per PortAudio limitations, we can only set a `framesPerCallback` variable, and a total duration in frames, from the command line. The PortAudio `framesPerCallback` variable will typically be used to set the period size of the ALSA driver, but *not always*: the driver's period size could end up different (quantized to the nearest possible value). Furthermore, we decided on developing another test and visualization approach (implemented in a script, `collectmirq.sh`): running a user-space program with different period and buffer size settings, and recording and plotting only the occurrences of periodic functions (IRQ or timer functions) of the driver (a technique similar to the approach on Figs. G.3 to G.5). The results are visualized on a plot, which in this case, has the time axis as the ordinate (vertical), while the abscissa is used as an index of a particular test run acquisition. This allows for side-by-side comparison of runs, obtained with different software: with either `latency-mod.c` or `patest-duplex-wire.c` programs, using either `dummy-mod` or `hda-intel` drivers, and with different buffer and period sizes. Plots of such a side-by-side comparison are given on Figure G.20.

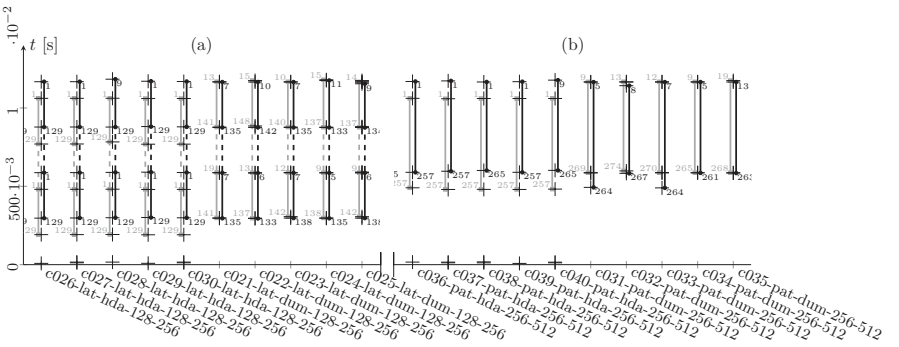


Fig. G.20: Rendering of data obtained using `collectmirq.sh`, displaying the periodic ALSA driver functions: a) 10 runs using the `latency-mod.c` program (period/buffer size 128/256 frames); and b) 10 runs using the `patest-duplex-wire.c` program (period/buffer size 256/512 frames); for different drivers (see text). Test duration in all cases is 512 frames.

The situation on Fig. G.20 made us aware that the *phase* between the different directions (playback or capture) of the drivers' periodic functions (the interrupts, or the timer functions) may be relevant to our problem. For each combination of: a user-space program (`latency-mod.c` or `patest-duplex-wire.c`); period/buffer size setting; and a given driver (`dummy-mod` or `hda-intel`) - the `collectmirq.sh` script will perform 10 debug log acquisition runs (for a total of 200 test runs). Acquisitions are numbered per sequential user-space program

runs, and this is the starting number on the abscissa tick labels on Fig. G.20. On the plots on Fig. G.20, the + markers indicate the time moments when the given periodic functions have been logged. In principle, we could have deduced the stream direction (playback or capture) of a periodic function, by printing out the driver structure variable `substream->stream` when the function runs; however, we wanted to keep the driver timing interference due to printouts to a minimum (the timestamp is obtained from the periodic function entry in the `ftrace` log), and since we already print the `.pointer` values, the code deduces whether a periodic function is for playback or capture, by parsing the data printed by the `.pointer` function called in the context of the periodic function. However, the `hda-intel` driver may start operating with `azx_interrupt()` calls wherein `.pointer` does not run - which is the reason for the "lone" + markers on Fig. G.20. For all other markers where the stream direction is known, we plot additional elements: for playback, to the left a square node and `.pointer` value, in gray; and for capture, to the right a circle node and `.pointer` value, in black (note that the value numbers may be truncated towards the edges of the plot). To ease the visualization of periodic behavior in time, the nodes for a respective direction are connected with vertical lines of respective color – and if applicable, every second such line is drawn dashed.

On the plot on Fig. G.20 a), where the drivers are set to period/buffer size of 128/256 frames, it is visible that the left half (showing `hda-intel`'s operation), generally reports either 129 or 1 as the `.pointer` value for either direction; conversely, the `dummy-mod` (right) half shows values like either 137 (138 ... 148) or 9 (12 ... 19) for the playback (gray); and either 133 (135 ... 141) or 5 (6 ... 10) for the capture (black). Something similar happens for period/buffer size of 256/512 frames, on the plot on Fig. G.20 b): the `hda-intel` left half reports either 257 (265) or 1 (9) for either direction; while the `dummy-mod` right half shows values like either 265 (268 ... 274) or 9 (12 ... 19) for the playback (gray); and either 261 (264, 267) or 5 (7 ... 13) for the capture `.pointer` value (black). Clearly, the `.pointer` values tend to indicate the period size boundaries in the buffer; for `hda-intel` they are more stable – while the variance in the `.pointer` values for the `dummy-mod` driver can be explained by the fact that here the value is calculated based on elapsed time in the timer functions; and due to preemption, we cannot expect that these functions will execute *exactly* periodically (or at least, as exactly as an IRQ handler, responding to an interrupt by a dedicated hardware timing signal, would).

However, there is another fundamental difference between the `hda-intel` and `dummy` plot halves on Fig. G.20: for the `dummy` driver (and ignoring the jittering due to preemption), it seems that both playback and capture timer functions tend to execute very close to each other (nearly "at the same time"), as it would be expected for linked streams. However, for the `hda-intel` driver, it seems that the execution of periodic interrupt handlers for one of the streams – the playback one (gray) – is consistently occurring earlier from the other one, which we can interpret as a phase offset/shift; and in spite of this, the

`.pointer` values reported still match across directions (that is, both directions will report a `.pointer` value of, say, 129; even if the interrupt for one executed significantly earlier than the other)! Comparing the a) and b) sides on Fig. G.20, it seems that this behavior of the `hda-intel` driver is persistent across changes of buffer and period size settings. Further investigation revealed that for a period size > 64 frames, this phase delay/offset (or the difference between moments of time, where corresponding playback and capture IRQs execute) is consistently around 48 frames (1.088 ms at CD quality). It was this behavior of the `hda-intel` driver, that we aimed to simulate in the `dummy-mod` driver: we set up the timer functions, such that the playback one would be consistently delayed early for 48 frames from the capture one; and we set up the `.pointer` function to return quantized values based on the elapsed time, which stabilized them. This resulted in a version of `dummy-mod` driver, called `dummy-fix`; a comparison (like on Fig. G.20) between the `dummy-fix` and `hda-intel` drivers' operation is given on Figure G.21.

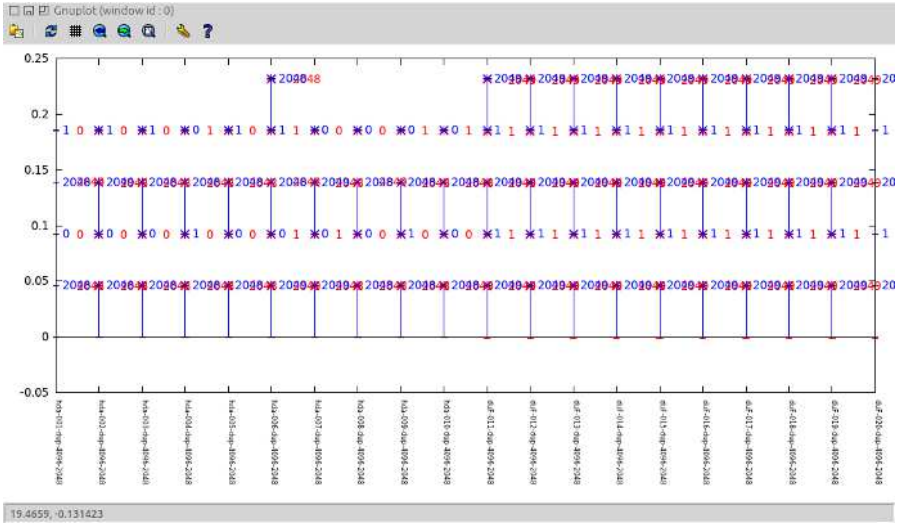


Fig. G.21: Screenshot of `gnuplot` (data obtained using `collectmirq.sh`), rendering the periodic ALSA driver functions, using `audacity` as user-space program: 10 runs using the `hda-intel` (left), and 10 runs using the `dummy-fix` (right) driver; period/buffer size is 2048/4096 frames in all cases.

Figure G.21 shows the typical usage of `gnuplot` with data from `collectmirq.sh` during our development: the plot layout is nearly the same as on Fig. G.20, except here the playback direction is colored red, and the capture one blue. Again, it shows a comparison of timing and values of pointer positions, but this time during CD-quality full-duplex operation of Audacity (whose underlying PortAudio tends to choose period/buffer size of 2048/4096 frames for either of the drivers). While not very visible on this scale; the playback pointer position of the `dummy` driver is delayed 48 frames early from the capture one

- simulating the behavior of the `hda-intel`; and in that, both drivers behave relatively similarly (as there is not much difference as we traverse the "takes" along the x-axis). This early delay of playback pointer position, and the quantization of pointer values, is what "fixes" the virtual `dummy-mod` driver, so it can perform a CD-quality, full-duplex operation in Audacity – *without* a drop like the one shown on Fig. G.11.

Unfortunately, currently we cannot say exactly *why* this kind of a change of the virtual driver's behavior, would have an influence on the detection of a problematic audio performance in PortAudio, which is otherwise unnoticed by ALSA itself. The delay (or phase difference) between playback and capture IRQs could be an inherent behavior of the Intel HDA controller hardware, or it could be due to a mode of operation set specifically by this version of the Linux `hda-intel` driver. The hardware IRQ timing behavior isn't specifically addressed in Intel HDA documentation like [41] or [44] (which handles architectural topics, that are independent of OS and driver choice). The playback/capture delay behavior of `hda-intel` could also be one of several hardware modes, chosen due to a special setting enforced by the driver - in which case, the question of why would the `hda-intel` driver developers choose to implement it, arises; it is possible it could have been a workaround for a bug. Indeed, after reporting our fix on [17], we were informed that our PortAudio drop problem, may in fact have been triggered by a PortAudio bug [45], that was fixed only recently (in 2013). An answer could probably be found by deeper review of related posts in `alsa-devel` mailing list, and the history of HDA Linux driver files (as found in the official `git` repository of `alsa-driver`). Unfortunately, a full clarification of this issue would require a more detailed investigation, than what we could afford in this project, due to time constraints. All of the code used in the development, discussed in this section, has been released through [1] (look up the `alsa-tests` directory) – or alternately, it can be found, differently organized, via the mailing list thread [17].

G.6 Profiling the CD-quality, full-duplex operation of FTDI FT232RL USB-serial IC

Having successfully solved the issue of a virtual CD-quality full-duplex ALSA driver, capable of writing data in the capture stream (in the form of the `dummy-fix` driver described in the previous section), we were persuaded that it would take nothing more than replicating that mechanism – where the playback timer function is delayed 48 frames early in respect to the capture timer function, and the `.pointer` function returns frames calculated based on elapsed time, which are quantized – in our AudioArduino driver, so that we could demonstrate full-duplex CD-quality operation with actual hardware; in this case, an Arduino Duemilanove.

By this time, however, we were already aware of the major deficiencies of our approach, that became apparent in the course of this project. In re-

view, our new work would be based on the full-duplex, 8 bit/44.1 kHz/mono AudioArduino driver [3a], which keeps the original `ftdi-sio` USB serial driver (for the FTDI FT232RL USB-serial chip on the Duemilanove), and adds an ALSA interface on top, based on kernel timer functions (inherited from the ALSA `dummy` driver). As mentioned in Sect. G.4, the `ftdi-sio` driver allows for $f_B = 200000\,\mathrm{B/s}$ data rate throughput through the USB-serial chip; while uncompressed CD-quality audio demands a throughput of $f_{aB} = 176400\,\mathrm{kB/s}$. On one hand, the fact that $f_{aB} < f_B$ should indicate that such a driver could be expected to function properly; on the other hand, the inequality G.4.10 does not hold for these settings, so it is likely that the driver could be susceptible to timing errors due to kernel preemption. However, we expected that by moving to high-resolution kernel timers - and otherwise implementing the architecture of `dummy-fix` - would minimize these effects; so as to result with a driver which, at least most of the time, reliably operates in full-duplex mode. Then again, we are aware from subsection G.5.3 that an ALSA driver architecture based on timer functions - even the high-resolution kernel ones - cannot fully replicate the DMA and IRQ soundcard mechanism, that ALSA is primarily meant to address. This would be even more true in this case: while USB does make use of DMA on a low level (as USB controllers are attached to the PCI bus), this is in a sense abstracted for us - as we interface with relatively high-level functions of the `ftdi-sio` driver, which execute before (or after) any potential DMA operation would occur, and may already include the effects of any IRQs that may have executed in relation to the data transfer operation. In this sense, here we are even further from the DMA+IRQ model, than when we merely attempted to simulate it in a virtual driver – and this, in itself, should alert one to expect potential problems in the intended operation of the planned driver.

As these concerns couldn't help us *a priori* predict the behavior of a CD-quality AudioArduino driver, there was nothing else left but to try it. Before we address the problems that we encountered during its development, let's first note some specifics of USB communication.

G.6.1 A closer look at USB and full-duplex

The 650-page USB 2.0 Specification [46] defines three USB data rates in terms of signaling bit rates: high-speed at 480 Mbit/s, full-speed at 12 Mbit/s, and low-speed at 1.5 Mbit/s; when the Arduino's FT232 is attached to our development PC, the kernel first recognizes it as a "new full speed USB device using uhci_hcd" in the `syslog` - which means that USB data between the PC and the FT232 is exchanged at 12 Mbit/s. Then, the specification defines the USB cable as containing four wires: V_{BUS} (nominally at 5 V), GND (ground at 0 V), and D+ and D- (data+ and data-, also emphasized on Fig. G.26). It is notable that the data pins D+ and D- form a *differential* pair: they can be seen as carrying the same data, but with opposite phase (or, as parts of the very same wire that carries a current signal, that has been turned in the opposite direction at one end). This means that USB is incapable of physical full-duplex,

understood as existence of two different data signals at the same time (as in the case of RS-232 serial) - simply because there is no other physical medium (like a wire) in the cable, that would carry a second signal. At the same time, this implies that electrically, the USB cable data wires represent a *bidirectional* bus: the USB devices must be able to take turns in sending data over the medium of D+ and D- wires, which means they must be able to terminate their end of the connection with high or low impedance, as appropriate. The USB 2.0 Specification also defines a tiered star bus topology, where tier 1 is the USB Host (and notably "*there is only one host in any USB system. The USB interface to the host computer system is referred to as the Host Controller* [46]"); while the other tiers are populated by USB Devices, which are then categorized into: USB "Hubs"; and devices implementing "Functions" (such as a soundcard, or a USB-serial chip). USB Devices expose a set of endpoints, with which the host forms (logical) pipes (which can be of stream or message type); endpoint 0 is reserved for control messages over the "Default Control Pipe", which a device must implement, so it exists once the device is powered on the bus. USB recognizes four types of data flow over a logical pipe: Control, Bulk Data, Interrupt Data, and Isochronous Data Transfers. Using the `lsusb` program under Linux, we can see that the FT232 exposes two USB endpoints: EP 1 IN (with endpoint address 0x81) and EP 2 OUT (with endpoint address 0x02), both of which are of Bulk transfer type, and have a maximum packet size of 64 bytes.

Most importantly, the specification defines USB as a polled bus (meaning that devices are asked in turn if they have any data to transmit), where access of the devices to the interconnect is scheduled in advance, and where the scheduling is controlled by the host - in that the "*Host Controller initiates all data transfers* [46]". In relation to this, the USB spec defines a frame, which (unrelated to the concept of an ALSA frame) simply represents a time base/interval of 1 ms established on full- and low-speed buses; for high-speed USB, an additional time base of 125 μs is defined, called a microframe. We may have met the concept of scheduling already: within a kernel context, we perform scheduling any time we use a function like the aforementioned `add_timer()` of the Linux standard timer API, even in context of periodic functions - because that function, simply, allows us to schedule the execution of some function at some time in the future. In the context of USB, something similar occurs - except that the host controller needs to manage the scheduling of messages not just for one, but for *all* devices that, via hot-plugging, may appear on (or disappear from) the USB bus. All of these factors represent a significant departure from the concept of full-duplex, we may be acquainted with from RS232 serial communication; thus, we have attempted to illustrate how some of these factors in USB communication may have an influence on our use case, on Figure G.22.

In absence of a USB analyzer device, which could have provided a detailed timing of each USB packet on the bus, the next best available thing is to capture and analyze the low-level kernel perspective on USB traffic: the mes-

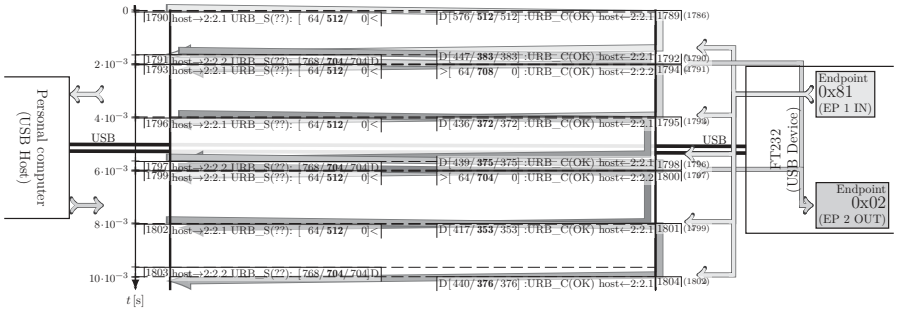


Fig. G.22: Visualization of a snippet of URB requests and responses, representing USB traffic; captured using `usbmon` / `tshark` on Linux during a full-duplex, CD-quality operation of Audacity with Arduino's FT232

sages, which the kernel exchanges with the USB host controller device on a PC, known as USB Request Blocks (URB). This is not a term defined in the USB specification; and on the official website, it seems to be mentioned only in presentations by Microsoft employees (e.g. [47]) - but is also used in the Linux kernel. Under Linux, capturing the URBs is possible using applications like `tshark` or Wireshark (and by proxy, the kernel debugging facility called `usbmon`; see also sect. G.7), and results with a timestamped list or a log, which may be difficult to parse visually at first glance; Figure G.22 visualizes the same information a bit differently. To begin with, all the packets are captured on the PC, and so we can consider the timestamps of the captured URBs to be in the PC clock domain; therefore the time axis (ordinate pointing downwards) is drawn from the PC side. The PC communicates with USB endpoints on the device, which have a specified direction and a four-bit number, encoded in the endpoint address: e.g. the hexadecimal endpoint address 0x81 is 10000001_2 in binary; the lowest bits 0 to 3 (0001_2) specify the endpoint number 1, and the most significant bit 7 (1_2) specifies the direction IN. Note that the direction is given from the perspective of the USB host, not the device (which actually contains the endpoints): when the host wants to read data from the device, it initiates a request to the IN endpoint (which transfers to host); and when the host wants to write to the device, it initiates a request to the OUT endpoint (which transfers from host). These requests, and the responses from the device, are in the form of URB messages. Note that as per the USB spec bus topology, there may be a USB hub in between the PC host controller and the USB device (here seen as a USB "Function", as opposed to a hub); that detail is ignored on Fig. G.22.

On Fig. G.22, text labels with some information headings from the captured URBs are plotted, aligned vertically to their timestamp position on the time axis. The text labels are aligned horizontally in respect to their direction (and otherwise adjusted to avoid overlap): the USB host can only issue requests

(considered as being of a type called `URB_SUBMIT` by `usbmon` / `tshark`), and the USB device can only issue responses to those requests (which are classified as a type called `URB_COMPLETE`). The submit URBs are aligned at the left, and the complete URBs at the right, inside the central diagram on Fig. G.22 (bound by two vertical lines). The text content of the labels is "mirrored" on each side, and it contains: a direction string (like "host → 2:2.1", where "2:2.1" stands for bus number 2, device number 2, endpoint number 1); a type and return status string (like "URB_S(??)", where we have replaced "??" for the status reported as "-115 (-EINPROGRESS) Operation now in progress" by `tshark`); a sizes/lengths string (like [64/**512**/0], where the first number is actual size of the captured URB; the boldfaced second is so-called URB length which is either a request for a given amount of data, or identical to the data payload length; and the third number is the length of the data payload, which can be zero); and a direction character (either '>' or '<' if the data length is 0; and 'D', which is our rename of the '0' which is reported by `tshark` when there is data available). Note that on Fig. G.22, all submit URBs have a -115 status (in progress), and all complete URBs have a status of 0 (OK).

Each URB is numbered, which is shown on a label on the respective outer side of the central diagram; the figure shows a snippet of URBs from number 1789 (whose timestamp is also taken to represent the starting time 0 of the axis) to number 1804 in that particular acquisition. Software like Wireshark can automatically parse information on, which URB "complete" is a response to which URB "submit" request - and provide hyperlinks in the GUI for easy navigation between the two; this can be exported as a "request in" number, which for the response ("complete") URBs is shown in parentheses to the right of the respective URB number on the figure. We have attempted to additionally visualize this relationship, using the thick "trapezoidal" lines behind the labels in the central diagram on Fig. G.22. To address legibility, these lines generally become darker in shade as time progresses - but also, every second one has a sudden change in shade, as well as a different end position at the right side. The trapezoidal shape should indicate (not to scale) that a URB would spend relatively short time traveling on wire, and most of the elapsed time between a request and response is spent waiting for a response to be generated by the device. So we can read the diagram as follows: with URB 1790, the PC host submits a request to EP 1 IN of the device, for 512 bytes ([64/**512**/0]) to be delivered to the host (read from device), in a URB 64 bytes in size. After some time, the host sends another request, URB #1791, to EP 2 OUT of the device, this time asking the device to accept 704 bytes ([768/**704**/704]) from the host (write to device); the data is sent along with the request, making the total size of the URB request 768 bytes. Soon afterwards, URB "complete" number 1792 comes in from the device, which is a response to URB #1790; in response to request for 512 bytes, it carries only 383 bytes ([447/**383**/383]) in a URB 447 bytes long. This is followed by another request submit by the PC, in URB #1793, for 512 bytes ([64/**512**/0]); and soon afterward URB #1794 comes in from the device, which is a response to URB #1791, apparently acknowledging

that the device received 708 bytes ([64/**708**/0]) in a URB without data (64 bytes long). These events seem to be repeated with some regularity at a period of 4 ms, which we can relate to the 1 ms "USB frame" time base - but also to the duration of a jiffy on our development platforms. It is notable that it takes far less time for the device to respond to a write request from the host (e.g. URBs #1791-#1794), then it does for it to respond to a read request (e.g. #1790-#1792); an obvious reason is that the FT232 device would have to wait for data to come in from the serial side, before sending it to the host over USB. Furthermore, the PC host seems to consistently ask to read 512 bytes, or write 704 bytes, of data per request; however, in URB #1794, the device acknowledges receiving 708 bytes, while the write request in #1791 carried only 704 bytes; arguably, this could occur if at a previous time, the device could not have accepted or acknowledged the entire payload written by the host, and so in #1794 it would acknowledge the missing piece it handled in the meantime. Also, the device may send back the entirety of the requested 512 bytes per read (#1789), but it may also send less than that (#1792, #1795, #1798, #1801). Since the "complete" URBs are the responses, and ultimately the confirmations of a successful read or write transfer, we have their positions for another set of shaded direction lines, to the right of the central diagram on Fig. G.22. The shadings and directions of these lines are related to the respective endpoint on the device: thus URBs #1794 and #1800, as responses to a write from the host, are sources of lines towards EP 2 OUT – and all the others are destinations of lines originating from EP 1 IN. The same kind of direction lines are also repeated near the PC box, for reference.

Ultimately, Fig. G.22 reaffirms the half-duplex nature of USB: the requests from the host, and the responses from the device, cannot possibly travel over the single differential pair D+ and D- in the USB cable at the same time, in full-duplex fashion. The salient point, however, is that since the USB packets are exchanged at a much higher bitrate (12 Mbit/s for full-speed USB) than the target serial 2 Mbit/s bitrate, we obtain not just the impression of, but actual full-duplex operation over serial, as witnessed by the RX and TX signal acquisition on Fig. G.25. On the other hand, we can also expect that kernel operation on this level will also be susceptible to jitter - independently; which may compound with the jitter effects we already recognized as part of the use of kernel timer function within ALSA drivers.

G.6.2 An elusive overrun error, and the FT232 FIFO buffers

As mentioned in the start of this section, our efforts in this phase were focused on implementing a CD-quality version of the AudioArduino driver - which we called `snd_ftdi_audard-an16s`; however, testing its final versions demonstrated a specific problem, which is shown on Figure G.23.

The screenshot on Fig. G.23 depicts Audacity during a digital duplex loopback test [3a], where we program the Arduino to simply resend each byte that it received as part of an audio playback operation; therefore, in case of

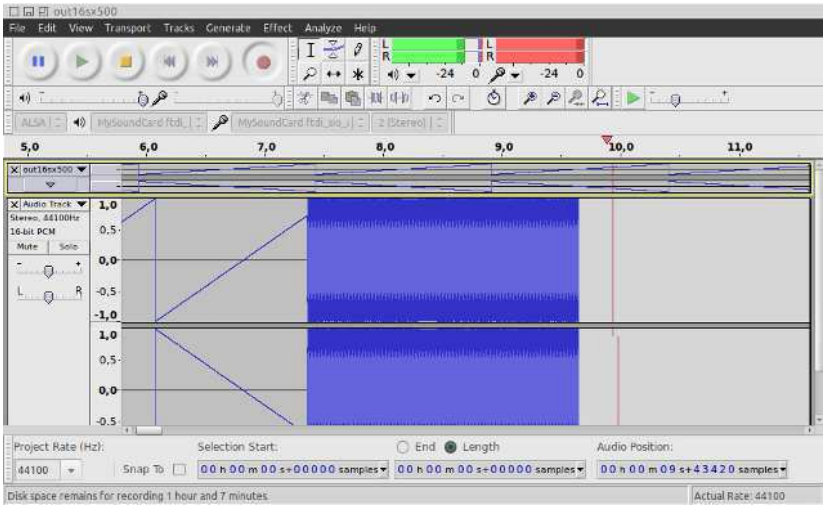


Fig. G.23: Screenshot of Audacity, working in CD-quality full-duplex mode with an AudioArduino driver, and a Duemilanove as digital duplex loopback device. The outgoing/playback track is the first one (vertically collapsed), the incoming/capture track is on the bottom (expanded); note the error in the capture stream, starting right before the 7.5 s mark.

a proper full-duplex operation, the capture stream should be identical to the playback stream, and any other result would indicate either driver or hardware problems. As an input (playback) stream, we use a linear ramp of all the 2^{16} values of a 16-bit signed integer, in antiphase per channel (we use the `genbindata_sawramp_16s.pl` script, mentioned in Sect. G.5.2, to generate this input audio data); this takes approx 1.48 s to reproduce at CD-quality, and a loop of this input would thus represent a sawtooth wave with a fundamental frequency $f_0 \approx 0.67$ Hz - that is, an infrasound. As such, its role is not so much to allow for an auditory test, but instead to allow for easier visual identification of possible errors in the capture stream (and thus errors in the loopback duplex test).

Note that on Fig. G.23, for up to some 7.5 seconds of the operation, everything is fine; which means that we can exclude ALSA driver problems, such as circular buffer wrapping errors (since $7.5\,\text{s} \gg 0.046\,\text{s}$ of the typical period duration in Audacity, corresponding to period size of 2048 frames), as the cause of the duplex loopback error. However, it can be something of a challenge to state exactly what *is* the cause of the error: inspecting the debug mode output of PortAudio and ALSA during our tests, indicated that neither of these components report registering an xrun (as an indicator of timing errors on the kernel level). In any case, *had* we experienced an xrun, we would have expected either a period's worth of dropped frames (e.g. like on Fig. G.11), or a corresponding amount of silence; which is seemingly not the nature of the error on Fig. G.23. Furthermore, during our tests, the error could have appeared randomly as early

as 1-3 s, and as late as 30 s after the start of a full-duplex operation: that means that obtaining `ftrace` kernel debug logs may be unfeasible for these kind of tests (given the massive amount of data that would be generated on the order of tens of seconds), meaning that we would have to look for an alternate way to confirm the cause of this error.

By inspecting multiple errors of the sort, we observed that when the error in the capture stream of the kind on Fig. G.23 is zoomed in, it also resembles a sawtooth wave, except with a period of 256 frames (or 256 samples per channel); this is a non-linear distortion in respect to the original signal. Furthermore, occasionally the error takes form as a sudden inversion of the channels' data (i.e. the left channel starts showing the phase that was previously in the right channel, and vice versa). This points to an amount of bytes, which is not an integer multiple of the frame size (for CD-quality, 4 bytes), being "lost" or "dropped" somewhere in the round-trip from playback to capture – causing a framing error; such a situation is shown on Figure G.24.

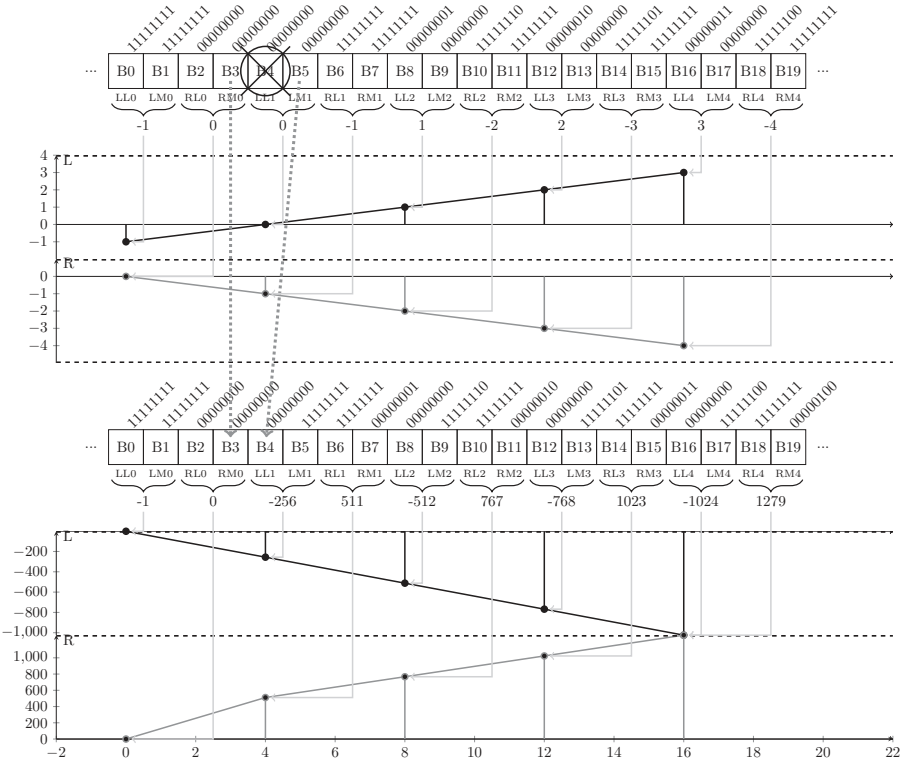


Fig. G.24: The effect of dropping a single byte from an interleaved CD-quality stream; byte array and stereo waveform plot of: original stream (top), stream with dropped byte (bottom).

On Fig. G.24, the abscissa can be thought of as an index of a byte, in the interleaved CD-quality stream; or alternately, time in units of a byte period (as per the audio rate) – although time delays due jitter are not represented. The "byte array" graphic representation is an alternate rendering of Fig. G.13, which shows the index of a byte inside a square, and the binary content shown rotated on top; below, we use labels where the first letter (L, R) indicates the channel, the second (L, M) indicates least (LSB) or most significant byte (MSB) weight in the 16-bit sample, and the third (0,1,...) is index of the frame; and below the braces, we show the signed integer decimal value of the 16-bit binary value of the sample. These decimal values are then plotted, with linear interpolation, on the plots below - as the waveform of the respective audio channels. The top byte array can be seen as the input (to the card/Arduino from the PC) - or the "original", playback data; the bottom can be seen as the returned output from the Arduino, that becomes the capture data in the PC.

Since our driver deals strictly with trafficking uncompressed audio stream data, there is no framing information whatsoever embedded in the signal, that could help the system identify that a framing error has occurred – all we have is the expectation that at these settings, an ALSA frame is 4 bytes wide, and frames are arranged sequentially one after another in the stream. That is why, the loss of a single byte at the fifth position (B4) on Fig. G.24 would cause the data in the playback stream to effectively be *shifted* in the capture stream - with complete disregard for the interleaved frame format, which has disastrous consequences: as visible on the figure, not only does the corresponding range (and thus the slope of the linear interpolation, which affects the frequency domain spectrum of the signal) increase significantly, but the phase of the signal changes as well! LSB data is now shifted to MSB position, which would explain the increase of frequency we observed in the error earlier (in a 16-bit linear ramp, the LSB values change 256 times more often than MSB ones). If there were two bytes dropped on Fig. G.24, say B4 and B5, then we would have observed not a distortion, but an exchange of the left and right channel data. If we dropped exactly 4 bytes, we would have dropped an entire frame, which - beyond the loss of the corresponding sample(s) in the channels - would not have caused any *further* distortion to the signal. Of course, the exact behavior of the distortion (when it occurs), would depend not only on the amount of dropped bytes (the size of the error), but also on where within a frame does the error start – and the characteristics of the input signal: the change in frequency we've observed here, is specific to the linear ramp nature of the input we used. Let us lastly note, that for the process opposite of the byte removal one (discussed so far) - the *insertion* of a byte in a stream - we would have observed the very same kind of distortion effects.

Awareness of this error mechanism does not reveal where the loss of bytes may have occurred, however. Eliminating driver operation (mostly the `dma_area` circular buffer wrapping mechanism) problems as a cause, we're left with the part of the route the data takes through USB and the Arduino, and back again. To confirm this, first we used the USB debug facility in the Linux ker-

nel called `usbmon` (see /Documentation/usb/usbmon.txt in the kernel source). It provides timestamped and encoded (but human-readable) strings similar to `ftrace`; and we used the command-line tool `tshark` (part of the Wireshark open-source analyzer software) to interface with `usbmon` and capture USB packets with their entire data payload. Finally we used a script to extract and concatenate payload data into binary files for each direction (playback and capture), which lent themselves to comparison with binary diff tools such as `dhex`. This procedure confirmed that the data in playback USB packets is identical to the original data, but the capture direction USB packets contain the error – which narrows down the list of suspects for the location of error origin to the Arduino board.

Next, we wanted to confirm what happens on the Arduino board itself – and the best opportunity for that was to capture the RX and TX serial signals between the FT232 and the ATmega328 chips (see Fig. G.6) on the board. To troubleshoot an error of this kind, we would have had to record the signals on the order of seconds, which along with multichannel sampling of binary signals, implies the use of a digital logic analyzer (as opposed to an oscilloscope; even if the distinction gets blurred in recent times). Here we used a Saleae Logic device, along with its accompanying software (also called Logic). The Logic software is not open-source; however, a freeware download is available, built natively for Linux (and only recently, we also learned that it is possible to use the open-source `sigrok` suite to interface with these devices [48]). Using the Logic device and software, we obtained acquisitions of the RX and TX signals during full duplex operations, one of which is shown on Figure G.25.

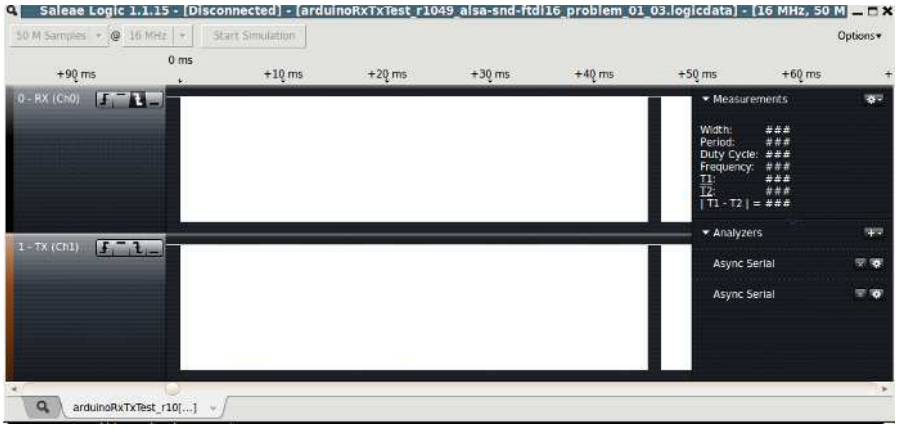


Fig. G.25: Screenshot of first 50 ms of Saleae Logic analyzer capture, measuring the TX and RX (serial) lines - connecting the FT232 USB-serial chip and the ATmega microcontroller, on the Arduino Duemilanove - during a CD-quality full-duplex loopback operation with Audacity.

While on this zoom level of Fig. G.25, the individual RS232 byte signal levels cannot be seen (as they are visually compressed) - a clear "silence" gap

is visible at approximately 46 ms (≈ 2048/44100), which corresponds to a typical period size (2048 frames) chosen by Audacity. Note the likeness to Fig. G.9: the gap is due to the USB rate (200000 B/s) being faster than the audio rate (176400 B/s, which controls the period time). In the version of the AudioArduino driver inspected on Fig. G.25, during playback there is a single call to the `ftdi_write()` driver function each period, with a period sized amount (here, 4096) of input bytes; this acquisition confirms that such a call does, ultimately, result with an uninterrupted serial stream out from (and back in to) the FTDI chip. Note that the Saleae Logic must sample at 16 MHz, to capture asynchronous serial (RS232) at 200000 bytes/sec, and be able to automatically decode it; and at that resolution, it can capture maximum about 1 second of data - which cannot help us capture and debug errors that might happen later in the full-duplex process (like on Figs. G.23 or (later) G.31). Therefore, we had to repeat test runs, until we would get a definite error within the first second of full-duplex operation, so that we could inspect the corresponding RX/TX signals: the Logic software allows export of decoded data from serial signals as an ASCII file, which then allows for easy comparison between the RX and TX signal contents. During this part of the development process, we have observed errors (recorded as phase distorted capture audio in Audacity, as on Fig. G.24) within the first second of full-duplex operation - while the corresponding Saleae Logic acquisition of both the TX and RX signals reported them to be both identical to the original input, and that therefore *no* data was being lost there; this indicates that the observed error must have happened somewhere in the FTDI USB/serial chip, with the data on its way from the chip to the PC.

The 43-page FT232 datasheet [49] notes that this chip has two on-board buffers, a 128 byte receive buffer and a 256 byte transmit buffer, placed between the USB serial interface engine and the UART controller within the chip. These are known as FIFO RX and FIFO TX Buffers, the designations being relative to the USB interface: "*Data sent from the USB host controller to the UART via the USB data OUT endpoint is stored in the FIFO RX (receive) buffer. … Data from the UART receive register is stored in the TX buffer* [49]". Or, in other words, data sent from the PC (the host) to the FTDI chip (via USB), ends up first in the FIFO RX buffer (the chip receives) – while in the opposite direction (the chip transmits), the FIFO TX buffer is used. These would be the main suspects for the location of the error: while the datasheet isn't explicit about this, we find it hard to imagine that these buffers would work in any other fashion, than as circular (ring) buffers - which means that they would be susceptible to, in particular, buffer underrun or overrun errors. At this point, it would be useful to have a better view of the placement of these buffers in the system, as well as a speculative model on the appearance of underrun errors in the buffers; this is what Figure G.26 attempts to illustrate.

While the block diagram on Fig. G.26 shows a more detailed context of the FT232 USB-serial chip in our development system, it is still a simplified rendering of information given in the chip's datasheet [49] and the Arduino

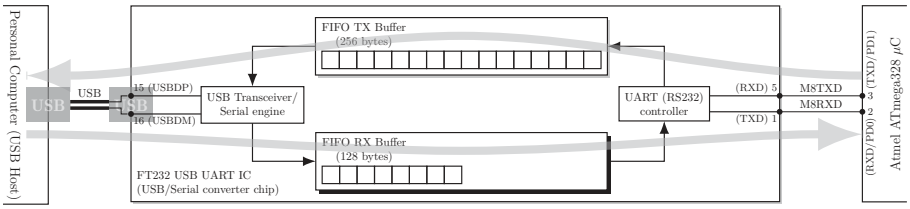


Fig. G.26: A representation of the buffers of the FT232 chip. The block diagram on top is a zoomed view of Fig. G.6, with the FT232 chip and its buffers emphasized (TX top, RX bottom).

Duemilanove schematic [50]. This block diagram preserves the same orientation as on Fig. G.6: the PC is to the left, and the ATmega328 microcontroller to the right, of the FT232 chip. The PC is connected through a USB connection, whose data wires are connected to pins 15 (USBDP) and 16 (USBDM), controlled internally by a USB transceiver engine in the chip. The microcontroller is connected through a serial connection, consisting of what the Arduino Duemilanove schematic refers to as RX and TX lines (M8RXD and M8TXD); note that these lines are named from the perspective (receive and transmit) of the microcontroller (they are connected, respectively, to: pin 2 (RXD) of the ATmega328, which is pin D0 on the Arduino board; and pin 3 (TXD) which is also pin D1 on the board) - and note that they are connected to the *opposite* named serial pins on the FT232 (respectively: pin 1 (TXD), and pin 5 (RXD)). These serial lines were the ones measured on Fig. G.25, and they are internally managed by the UART controller engine of the FT232 chip. The FIFO buffers, then, are placed between the USB Transceiver/Serial engine and the UART Controller engine of the FT232 chip, "halfway" between the PC and microcontroller endpoints.

The thick gray direction lines on Fig. G.26's block diagram show the route, that the data would take, in each direction. When the PC sends data, it goes through the USB connection - and after it is received by the USB Transceiver/Serial engine, the engine places it in the FIFO RX (chip receives) buffer. Thereafter, the UART controller reads data from the RX FIFO, and sends it, formatted as RS232 voltage signal, on its TXD pin – which, via the M8RXD line, arrives to the RXD pin of the ATmega microcontroller (lower gray line). In the opposite direction, the microcontroller sends data as RS232 formatted voltage on its TXD pin, which arrives at the RXD pin of the FT232, and is received by the UART Controller in the chip - which promptly stores it in the FIFO TX (chip transmits) buffer; the USB Transceiver eventually reads the data from this TX buffer, and sends it to the PC through the USB connection (when a corresponding URB request comes in; upper gray line on Fig. G.26). The FIFO buffers are represented graphically as rectangular arrays: since in printed format, subdividing the arrays at individual bytes would not be legible - we have chosen that one cell should represent 16 bytes (4 frames at CD quality) on

the diagram; the arrays are left-aligned for easier perception of the difference in size between the two buffers.

In order to account for a mechanism that results with a removal or an insertion of a byte in the stream (as on Fig. G.24), we will provide a speculative description of the evolution of the buffer's states in time, during a data transfer – consider Fig. G.27.

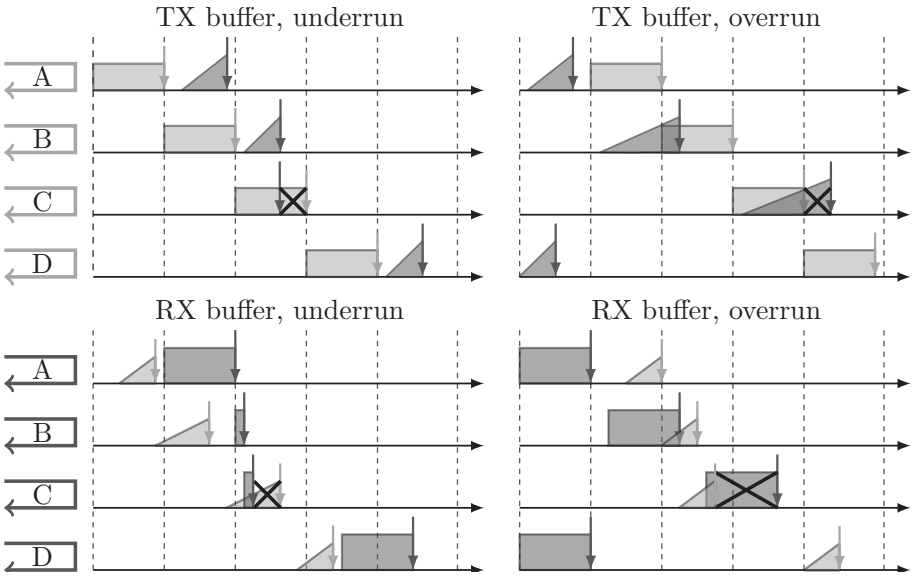


Fig. G.27: Representation of possible xrun states in TX (top) and RX (bottom) buffers.

Fig. G.27 visualizes the two types of "leaky bucket" errors, overrun and underrun, for each of the TX and RX buffers as on Fig. G.26. The axes can be seen as time, or position into the buffer; the buffers are assumed to be circular, so they wrap at the end of the axis. The "head" pointer variable, shown in dark gray (and taller), shows the state of writes into a buffer; the "tail", shown in light gray, shows the state of reads from a buffer, Each state (A, B, C, D) represents a short period of time, where the head or tail changed; the arrow in each stands for the state at the end of this period, and the duration between the states is arbitrary (not necessarily periodical). The shade of the arrows around the state labels at the left, indicates which variable is under control of the USB host; so for the TX buffer, the USB host controls the tail (reads), while for the RX buffer, it controls the head (writes). This is assumed to take a short time, so the corresponding shapes are rectangular. Conversely, the other variable in a state would be controlled by the serial UART, and can be assumed to change in a slower, incremental fashion, which is indicated by a triangular shape.

As example, for the RX buffer, the head increases when the PC has sent

data via USB, which ends up being stored by FT232's USB transceiver in the FIFO RX buffer (write); conversely, the tail increases when FT232's UART controller has read bytes from the FIFO RX buffer, and has successfully sent them on the M8RXD serial line. In this context, when reading from, occurs faster than writing into a buffer, inevitably a state is reached where there is no more data to be read, which is an underrun (the "bucket" leaks faster than it is filled). Conversely, when writing into, occurs faster than reading from a buffer, inevitably the buffer will run out of space, and will not be able to store incoming data, which is a state of overrun (the "bucket" is filled faster than it leaks). Thus, we can interpret, say, the TX underrun diagram (Fig. G.27, top left) as follows: in state A, head is ahead of tail, and everything is fine; after some time, in state B, the head has slowed down (it hasn't advanced as much as the tail), but there is still no error; later in state C, the head has stopped completely, while the tail has crossed it at the end of the state - bytes beyond the head are invalid (they represent previously processed values), and an underrun occurs; finally after some time in state D, the head has sufficiently advanced again, so the operation returns to normal. Note that in both overrun diagrams (Fig. G.27, right), the start is with state A where the head is to the left of the tail - this is meant to represent a valid state, where the head had just "wrapped" around the buffer boundary (which means that in absolute terms, at that point the tail is still behind the head).

What will the effect of the under/overrun (or *xrun*) be on the output stream, depends not only on the timing and sizes of the consecutive reads and writes - but also on how the engine is designed to handle the error. For instance, for an underrun, the engine might return only the valid bytes (so, less than requested); or it may return the entire request, however either returning previous values, or zeroes, as padding for the invalid values. For an overrun, the engine may either discard the incoming bytes that would have been stored ahead of the tail; or may choose to write them in, thereby overwriting values the tail hasn't read yet. In either of these cases, the engine may decide to reset one (or both) of the pointer variables to 0 (the start of the buffer). However, it seems that an overrun is a more serious error, as in the case of an underrun, the engine might simply stall the tail until the head advances; for an overrun, bytes always end up corrupted (we'd need an additional buffer where all incoming values would be previously stored, if we're to stall the writes in this buffer effectively). For instance, an overrun where the incoming bytes that would lie beyond the tail are discarded, where the tail and head are reset to the start of the buffer, and the tail is stalled until the head sufficiently advances, would account for loss of bytes in a stream as on Fig. G.24.

This model might provide a basis for relating a mechanism as on Fig. G.24, to the problem as observed on Fig. G.23; unfortunately we cannot confirm what is the exact cause of such errors. To begin with, the FT232 datasheet [49] doesn't even mention the words "underrun" or "overrun" (or "overflow" - even if it is typically applied in a different context). Some more information can be found in the source code files of the `ftdi-sio` driver in the Linux kernel – but be-

fore we address that, let us notice that on Fig. G.22 (which otherwise visualizes USB traffic during CD-quality operation of the same `snd_ftdi_audard-an16s` driver discussed here), the exchanged URB messages have a size of 64 bytes, when they do not carry any data (in either direction). This portion encodes information like URB type, status, etc.; and can be seen as the header of a URB message - covering the bytes with zero-based index 0 to 63 (0x00 to 0x3f in hexadecimal) in the message. But if there is any data carried with the URB message, then the first two bytes of the data section - the bytes with zero-based index 64 to 65 (0x40 to 0x41) from the URB header (and repeatedly at 64 bytes in the URB packet) - in fact represent *status* bytes, which contain information not otherwise present in the URB header. The `ftdi-sio` Linux driver, in fact, skips these bytes over (in case of a read from the device) before handing over the rest of the data to the kernel (including the ALSA component, in case of AudioArduino). Notably, these status bytes are *not* mentioned in the datasheet [49]; however, there is an application note from FTDI [51], concerning data throughput and latency of these chips, which does document their existence – but unfortunately, does not document their meaning, or the use of their values.

The only reference we've found so far, that documents the meaning of these status bytes, is the `C` header file `ftdi_sio.h` - part of the `ftdi_sio` driver in the Linux kernel; there we find the bit field information, of which we provide a verbatim copy in listing G.6.2:1.

```
* Byte 0: Modem Status                          * Byte 1: Line Status
*                                               *
* Offset       Description                      * Offset       Description
* B0   Reserved - must be 1                     * B0   Data Ready (DR)
* B1   Reserved - must be 0                     * B1   Overrun Error (OE)
* B2   Reserved - must be 0                     * B2   Parity Error (PE)
* B3   Reserved - must be 0                     * B3   Framing Error (FE)
* B4   Clear to Send (CTS)                      * B4   Break Interrupt (BI)
* B5   Data Set Ready (DSR)                     * B5   Transmitter Holding Register (THRE)
* B6   Ring Indicator (RI)                      * B6   Transmitter Empty (TEMT)
* B7   Receive Line Signal Detect (RLSD)       * B7   Error in RCVR FIFO
*                                               *
```

Listing G.6.2:1: Bit field map, as found in `ftdi_sio.h` - with meanings of individual bit flags of the status bytes, returned by the FT232 chip.

We have attempted to look further for official documentation mentioning this information, but in vain: for instance, an Internet search for "Transmitter Holding Register" limited to the manufacturers website, returned zero results as of 2014. At least, through listing G.6.2:1 we learn that the device itself can acknowledge an overrun error, through the bit B1 in the Line Status byte (byte 1). Just after the snippet on listing G.6.2:1, the driver code defines bitmask values for reading these bits, among them `FTDI_RS_OE` intended for the overrun error bit. This made us look elsewhere in the `ftdi_sio` driver code, to see if these values are used, and indeed they are - precisely before the spot where the status bytes are discarded, and the rest of the payload data is

transmitted to the kernel; in particular, `FTDI_RS_OE` is used in the `ftdi_sio.c` driver file, at the point in the `ftdi_process_packet()` function, shown on the stanza on listing G.6.2:2.

Listing G.6.2:2: Location, where overrun (`FTDI_RS_OE`) is checked in the second status byte (`packet[1]`) from the FT232 chip, in the `ftdi_sio.c` driver source file.

```
flag = TTY_NORMAL;
if (packet[1] & FTDI_RS_OE) {
        flag = TTY_OVERRUN;
        printk("ftdi: OVERRUN error  len: %d; packet: 0x%02x 0x
            %02x \n",  len, packet[0], packet[1]);
}
```

As it can be seen on listing G.6.2:2, the overrun check is simply used to set a flag variable to an error value; later in the code, this flag is used to determine whether to proceed with writing the data further to the serial port abstraction in user space. In our AudioArduino modification, we have always chosen to write the bytes to the ALSA part of the driver regardless of this flag, in order to have a more direct exposure to errors, should they occur. In fact, the original driver didn't even report these errors anywhere - which is why we have added the `printk()` statement in that part of the code on listing G.6.2:2. With this in place, we were able to rerun our tests, while monitoring for the report of precisely this error in the system log. We could confirm that there indeed is correlation between the occurrences of phase distortions in Audacity (as on Fig. G.23), and the printouts of the above overrun error message. Although, at times, we have also seen an error printout occur, without a corresponding distortion in the signal - we believe that can be accounted for, by assuming that at those times, the FIFO overruns that occurred are an integer multiple of the ALSA frame (4 bytes) in size. The interesting thing is that we used to get two distinct status byte value pairs printed, when an overrun error occurred: 0x01 0x02, and 0x01 0x62; the appearance of one or the other pair, didn't seem to be related to whether long-running distortion occurred in the signal at the same time. With the information in listing G.6.2:1, we can look more closely at what those values stand for: in particular, when the second byte's value is $0x02 = 00000010_2$, that means that all bits in the field are zero except for B1, which is the Overrun Error; and when its value is $0x62 = 01100010_2$, bits B1, B5 (Transmitter Holding Register) and B6 (Transmitter Empty) are set.

Unfortunately, this does not bring us to much further insight; to begin with, without further documentation, we can never be sure that the "Transmitter" - as used in the names of B5 and B6 bits - specifically refers to the FIFO TX buffer of the FT232 chip (as on Fig. G.26). If it does, we could speculate that the Line Status byte value of 0x62 signals an overrun in the FIFO TX buffer, and the value of 0x02 notifies of overrun in the FIFO RX buffer on the chip - but again, without an explicit reference in an official documentation, we cannot really claim that this is the case. In our tests, at times we have observed

that the 0x02 value occurs more frequently than 0x62, when an error appears. Because of this, we ran a few half-duplex tests - with only the Arduino writing to the PC, and the PC capturing only, without playback - where we observed only the 0x62 line status value occur; this should, apparently, confirm that 0x62 indeed signals overrun in the FIFO TX. In full-duplex context, we can observe that - besides the FIFO TX being twice as large as the FIFO RX buffer - the Arduino can only write in the TX buffer, that data which it received from the RX one; and from [3a] we know that the Arduino will copy a byte in mere $\approx 6.9\,\mu$s, with a jitter far less than that. As another example: in one of our full-duplex captures, a 0x02 overrun was signaled, resulting with a state as on Fig. G.11; comparison of USB URB payloads on the PC revealed that the write data (from PC to USB chip) is intact, but the read data (from USB chip to PC) is missing a continuous block of 37 bytes – while a comparison between serial RX and TX signals on the Arduino (as on Fig. G.25) revealed the full original data present on both. As per Fig. G.26, this could only be explained by an overrun in the TX buffer – but then, the interpretation that 0x62 signals a TX overrun and 0x02 signals an RX overrun is not correct.

Unfortunately, as mentioned earlier, we cannot claim what exactly happens, as we cannot state with certainty, what do the bit field names (in the Line Status byte bits) refer to exactly. The difficulty in tracking down this information was clarified, when we eventually stumbled upon a discussion in the `usb-devel` mailing list [52]; below, we provide extracts of the most salient points in this e-mail thread:

> “
>
> > *– Can anyone confirm that the state of the chip's TX buffer (at least whether it is empty) can be retrieved from the chip? There is no publicly available datasheet about this kind of hardware details.*
>
> *I don't have the datasheet, so I can't answer this for you, sorry.*
>
> > *– Is tcdrain() supposed to ensure that all TX data has been sent for any serial device? If so, why is it not implemented for this device?*
>
> *No one has implemented it to do so. Other usb−serial drivers have implemented this functionality, just not this one. Perhaps because no one has noticed before, or maybe the chip really can't do it.*
>
> *Without the specs for the device itself, I can't really tell for sure. If you do get ahold of the specs, let me know, and I'll be glad to work to add this support to the driver, as I agree, it is something that would be useful to a lot of people.*
>
> …
>
> > *I have e−mailed FTDI's support to ask whether the assumptions described above are true and whether there is no register to read the actual amount of data in the hardware buffer(s). Their website states you need*

> *to sign an NDA to get the register descriptions, but with some luck that isn't needed.*
>
> ...
>
> *OK, no luck. I got a reply that pretty much said "please sign this NDA" and since I'm not a lawyer, I'm not sure whether the conditions would allow me to use their datasheet for GPL code.* [52]"

Assuming that, in our case, the PC writes too fast to the FT232 chip, we too could have used something similar: if we could have retrieved the state of the FIFO RX buffer from the chip, we might have known in advance whether we're about to overrun the buffer – and we could have performed traffic shaping by delaying the writes; which may have helped avoid the FIFO RX overrun altogether. However, it is uncertain whether information obtained through a non-disclosure agreement (NDA) would be applicable to a project with open-source ambitions like this one.

As a last-ditch effort, we attempted to focus again on information which we can extract from the kernel, and perform an analysis, this time based solely on the amount of bytes written to and read from the FT232 chip by the driver. This approach ultimately failed in terms of bringing a solution; however, we provide an overview of it in the next subsection.

G.6.3 An inconclusive analysis - ftdi_profiler and visualization using multitrack_plot.py

With the conclusions from the previous subsection in mind, we decided to focus solely on the data exchange between the PC and the FT232 – implying that we would want a test, without the audio ALSA layer interfering with the process. This resulted with a new kernel driver: just like AudioArduino, it is implemented in a single header file (`ftdi_profiler.h`) which is utilized by the `ftdi_sio.c` source file of the official Linux `ftdi_sio` driver, modified in the same way as it was for AudioArduino; compiling it results with a driver kernel module `ftdi_profiler.ko`. As mentioned, it does not include any ALSA functionality - however, it includes the same organization of timer functions as found for playback (write to device) or capture (read from device) in the AudioArduino driver - and thus is hooked through the same two functions of the original `ftdi_sio` driver (`ftdi_process_packet()` and `ftdi_write()`). So instead of expecting interaction through ALSA or a serial port, the `ftdi_profiler` driver exposes files in the Linux `/proc` filesystem, which can be used to set: a period of time, an amount of bytes to write during that time period, and a duration time of the test; finally, there is a file `/proc/ftdi_profiler` exposed: reading from this file starts the process - a kernel timer function write loop (and the corresponding reception of data, like in a full-duplex AudioArduino operation) - if it hasn't been started already; if it has been started, the read from the file simply reports that the process is on-going. The process terminates itself once

the duration time, set previously through the `/proc` filesystem, has expired. Thus, there is no interaction with a user-space program during this process (aside from the start-up triggered by a read of the file in `/proc`).

Here we were mostly interested in performing debug data acquisitions of longer running operations - up to 30 seconds - to ensure that we would capture at least one buffer overrun per test run. Using the `ftrace` Linux kernel logging facility, in the full function graph mode like we used it previously (recall listing G.2:2b), would have generated immense amount of data; making it additionally difficult to process. Instead, here we opted simply for the driver printing a single line of information when the callback functions run in each direction (writing and reading), mostly focused on the timestamp and the amount of bytes processed, as well as printing of the FTDI status bytes in case of a read; however, we still used the `ftrace` engine here, in that we used its `trace_printk()` function - so we could avoid the overhead, associated with the usual `printk()` function of the Linux kernel. Of course, this is simply another version of a duplex loopback test, and it requires the Arduino connected to the PC, programmed to send back the bytes it received as soon as possible; the content of the bytes is however irrelevant here, as `ftdi_profiler` simply writes the requested amount of zero (0x00) bytes all the time.

With this test format, we didn't expect that the procedure would be too CPU intensive, so we made an attempt to capture the URB requests (as on Fig. G.22) during this test procedure as well, through the `usbmon` debug facility of the kernel. Unfortunately, the `ftrace` and `usbmon` debug subsystems, even if they both by design address the Linux kernel, are somewhat incompatible between each other, at least in the OS versions we worked with - for instance, we could not find a way to route a `usbmon` message to the `ftrace` log on the kernel level; so we attempted the most straightforward way - setting up a continuous read (via `cat`) of the `usbmon` tracing files, and redirecting that to the `ftrace` log external input file (`ftrace_marker`), from the shell. This introduces levels of indirection, that would make the `usbmon`- acquired data more unreliable; but in addition, we realized that the timestamps produced by `usbmon` are in a completely different format than those by `ftrace`! Simple attempts at correlation (starting at a common marker, and comparing the difference of time expired since then) between the two kinds of timestamps revealed that they are not predictably correlated. Because of that, we didn't use the data acquired by `usbmon` at the time, and the analysis below refers only to data produced by `ftdi_profiler` itself (however, since then, we learned of a method to convert usual kernel timestamps into `usbmon`-style timestamps [53], which would have assisted with time-correlating data from both domains more reliably). Still, the test procedure we described above, can be found implemented in a script we called `ftdi_profiler.sh` (available via [1]).

As in previously discussed tests, this procedure too generates a plain-text data acquisition log, which needs to be processed afterwards. The idea for analysis is essentially simple: try to find a condition on the PC, that would unambiguously trigger an appearance of an FT232 overrun; and prove it by

implementing an algorithm, that by processing the log-contained timestamp and byte amount information *only*, would "predict" the times of appearance of the overruns - which can be validated by the existence of records of actual overruns in the log. Of course, if we knew exactly what we were looking for, we could have written such an algorithm immediately; however, given that we were trying to identify such a condition in the first place, we again had to resort to customized visualization of data – and as the now famous quote (often attributed to J. Zawinski) goes: now we have two problems. Here the data is of a slightly different nature than in previous tests: the log mostly doesn't contain redundant data, and so the parsing can be a bit simpler. However, we have both timestamp position data, and integer values contained in multiple time-dependent variables, from which we would also attempt to derive new variables. In all, the most natural form of visualization of this kind of data to us seemed to be a waveform plot: time on the abscissa, and variable value on the ordinate; per variable. As this is a standard plotting approach, first we attempted to experiment with `gnuplot`: while it certainly doesn't have a problem with plotting this kind of data, we realized that one of the most important aspects in this case for us was the ability to quickly zoom in and out around data at particular points in time. While `gnuplot` has two interactive interfaces on Linux that allow this, the `x11` one is fast but difficult to work with (both visually and interactively), and the `wxt` one is rather usable but slow (especially in terms of zoom in and out) with the kind of data and plots we used.

Therefore, we looked into other possibilities for plotting and visual analysis of this data. We already had experience with Python's `matplotlib` from Sect. G.4.1; while we found this library very versatile and convenient to work with, we also found its response during zooming interaction rather slow. We then tried another plotting library for Python called `chaco` (v. 3.4.0), which we found to have a much faster response to zooming; unfortunately we hit a bug in that library, which unexpectedly started filling areas on the plot for large data (already at about some 110 k points). So we had to look further; and found a standalone GUI plotting application called `kst`. It is a part of the KDE desktop environment software, but we didn't have a problem finding a package that could run in our development OS, which uses the Gnome 2 desktop environment. Unfortunately, while the version we found, 2.0.7, could start up without a problem - it also caused a CPU hog (some 90% or more CPU utilization) as soon as a graphics session was started with our data loaded; this didn't get resolved even after we built the application from source, and as such this application version ended up being unusable to us. Eventually, we tried the Windows version of `kst` 2.0.7 running under the `Wine` emulator in Linux - and ironically enough, this version didn't suffer from the bug, so we ended up using this version. This application impressed us with its speed in terms of zooming interaction, and the allowance for limited manipulation data in the application using a specific syntax. However, the feeling of having settled down on a choice didn't last long: as we kept increasing the number of plotted vari-

ables we derived from the acquired data, `kst` kept on behaving more sluggishly; its screen can be separated into multiple plots, but they cannot be scrolled - so ultimately, we also ran out of space where we could meaningfully (in the visual sense) plot our variables; we experienced that more than three variable waveforms on the same plot become very difficult to read and analyze. But there was one more issue with `kst` which ultimately forced us to look further - which is the issue of data interpolation, shown on Figure G.28.

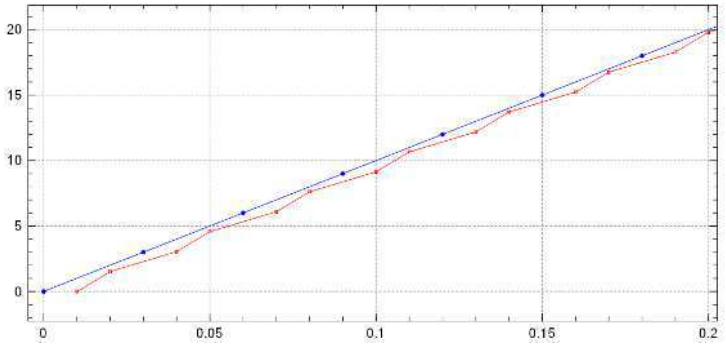


Fig. G.28: Result (lower, red, outlined squared marker) in `kst`, of our attempt to interpolate an input signal, defined at one set of time moments (upper, blue, filled circle marker) over time moments defined in another set

In `kst`, like in other plotting software, sampled data is handled so the time positions are usually defined as one vector, and the actual values of a signal at those positions as another vector; here "vector" being short for "data vector", having the meaning of one-dimensional array of real numeric values (integers or floating point numbers). Fig. G.28 is our attempt to resample the input values vector, defined at time moments $t_i = i \cdot \Delta t$, $i = 0, 3, 6, 9, ...$ (or $i = 3 \cdot n, n \in \mathbb{N}_0$), over the union with time moments at $t_j = j \cdot \Delta t$, $j = 1, 2, 4, 5, 7, ...$ (j being the "remainders" between i, or $j = n \cdot \mathrm{sgn}(n \mod 3)$ for $n \in \mathbb{N}_1, j > 0$), which may belong to another input vector. This means that the values stored in the vectors are in different time domains - or more properly, sampling domains. However, for a software like `kst` to be able to perform calculations on them (e.g. to subtract one signal from the other), it needs them in the same time domain - which then implies a need to resample the signals to a common time domain (the union of all time positions present in either of the signals). Therefore here, we would ultimately like to resample the input values, over the union of the time moments defined for both signals - which for the choices in this case, would boil down to all time moments $t_n = n \cdot \Delta t, n \in \mathbb{N}_0$. Fig. G.28 is the outcome, of what looked like the most straightforward approach for doing that to us: to enter in `kst`, as Equation, "[val_i (V2)]"; to enter, as X vector for the equation, "t_j (V3)"; and to check "Interpolate to highest resolution vector" (V2 and V3 are internal data vector names that `kst` assigns; val_i and t_j would be our custom names, which would be related to the indexing above). However, as

shown on Fig. G.28, the output is interpolated only over the t_j domain, and it doesn't include the time moments t_i; also, a proper linear interpolation would mean that the output values, as points, should lay exactly on the line formed by the input values - while here, clearly, the output values are slightly below it. We might consider this a documentation bug, as to our knowledge, the `kst` documentation is not explicit about how to do resampling of data values of this kind. We submitted a bug regarding this issue [54], although possibly at the wrong site (the Ubuntu bug tracker, instead of the upstream one for `kst`).

During our period of analysis with `kst`, we would have often been led to conclusions which seemed to be absurd - and becoming aware of the behavior on Fig. G.28 did finally account for some of that However, we also became aware that in terms of plotting, it may be that linear interpolation, as a technique, is not what we needed at all. Consider that our data at this stage consists of samples, taken at different time positions: the writes are periodic, as they are driven by a kernel timer function with a set period; the reads can also be seen as periodic, but with a different, independent period - given that they are driven by URB responses (Fig. G.22). As such, they generally occur at different time positions; or, again, they are in different sampling domains. Let us consider the case where we would look for the difference between two such signals, shown on Figure G.29.

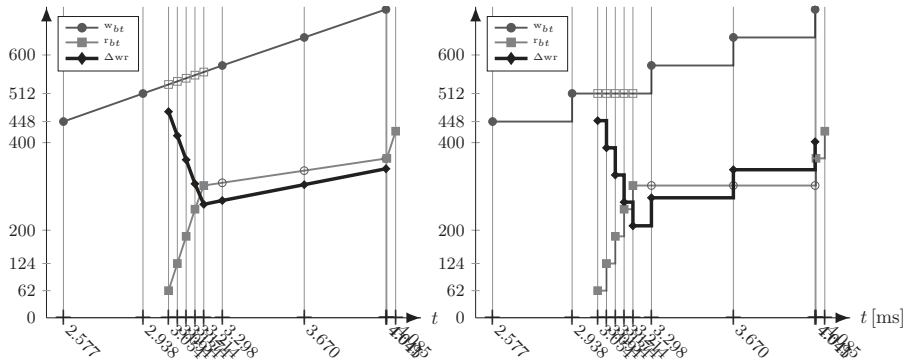


Fig. G.29: Linear (left) vs. null interpolation (right) plot for discrete signals in different sampling domains: w_{bt}, r_{bt}, and their difference Δwr (based on `ftdi_profiler` data acquisition)

Fig. G.29 shows two signals, based on a data acquisition from `ftdi_profiler` set to write 64 bytes at each period of 362.8 μs (which corresponds to the CD-quality data rate of 176400 B/s), but with a small modification. First, let's note that unlike tests like Fig. G.25, where we'd pass 4096 bytes at each call to `ftdi_write()` from the timer function - here we write only 64 bytes at a time; which is both the declared maximum packet size of the FT232's USB endpoints, and the half of the size of its FIFO RX buffer; so we would consider it a "short" write. The `ftdi_write()` function in itself basically allocates memory

for a URB request, copies the data received through the function's arguments to the URB request, and then queues the URB using `usb_submit_urb()`; the notable thing here is that the lower layers of the kernel USB implementation will decide when to physically send a URB on the USB bus - otherwise, the `ftdi_write()` doesn't block in waiting for a confirmation that the URB has been successfully sent, so it returns relatively quickly after being called. Since we've never experienced a problem with the `ftdi_write()` call itself, we felt justified in simply keeping a counter for the total amount of written bytes in the `ftdi_profiler` driver, and increasing it for the amount of bytes to be written as soon as `ftdi_write()` has returned - and using the value of this variable, which we've variously named `w(r)tot(b)` throughout code (or w_{bt} on Fig. G.29), as a metric for the total amount of written bytes. Thus, we can observe on Fig. G.29 that w_{bt} changes relatively regularly with the requested time period and amount of bytes. On the other hand, in the read direction, the `ftdi_profiler` driver layer gets the data only after the underlying `ftdi_process_packet()` function has extracted it; as such, in the variable we've named named `r(d)tot(b)` or r_{bt} we accumulate the actual amount of bytes received per call. Since it is not us that set the read period, but the lower layers of the USB framework, we might as well consider the read to be triggered like an interrupt; still, we can observe on Fig. G.29 that the read is also regular, albeit not strictly periodic: the variable increases in steps of 62 bytes (which is related to the existence of the status bytes in the read direction, as noted in [51]) in quick succession, and after $\approx 2\,\text{ms}$ that process repeats. Note the difference from the regime on Fig. G.22: there the reads and writes carry more variable amounts of bytes; for the short writes here, they are more consistent. Thus on a PC, when we inspect received data on the level of the kernel driver, we perceive short, 62 byte sized packets with a short period; when we inspect through `usbmon`, the same data (with status bytes) is concatenated in larger URB response packets, with a longer period. It is here, where we modified the data on Fig. G.29 for visualization reasons: in reality, in the quick succession the reads are only few microseconds apart, which would not be legible in a print format plot; therefore we artificially extended the time between the reads to $40\,\mu\text{s}$, so they could be resolved individually on Fig. G.29.

It is clear that on Fig. G.29, w_{bt} and r_{bt} belong to different sampling domains; on the abscissa, the + and the vertical lines mark the union of timestamps defined for both signals; the intersections of the vertical lines with the respective signals, where a timestamp is not already defined, are marked with unfilled markers - and represent where we must interpolate sample values, in order to properly define the difference $\Delta\text{wr}[t] = w_{bt}[t] - r_{bt}[t]$. The difference signal Δwr is only defined where a sample (interpolated or not) exists for both functions. It is here, where we can start perceiving the limitations of the linear interpolation plot: if the integer (i.e., digital) values were meant to be reproduced by a digital-to-analog converter and, say, a loudspeaker - then the linear interpolation (as on Fig. G.24) is entirely appropriate: since even if the DAC reproduces the values in a step fashion, the electric and mechanical filtering

involved will mean that in response, the loudspeaker membrane will move in a smoothed, almost linearized fashion. However, here we represent variables that only have meaning in digital operations - where it is important that once set, they keep their value until changed; which makes them more akin to a Heaviside step function when plotted. Resampling in a step function context, as opposed to linear interpolation, is often known as *null* interpolation. As example, imagine we're zoomed in a close range around the 3.670 ms mark on Fig. G.29 left, showing linear interpolation: the obvious deduction is that $\Delta\mathrm{wr}$ and r_{bt} both grow, and that $\Delta\mathrm{wr}[t] < \mathrm{r}_{bt}[t]$ - in the entire interval. However, if we'd look at the null-interpolation plot on Fig. G.29 right, zoomed in around the same range, the obvious deduction is different: here $\Delta\mathrm{wr}$ is constant - it does *not* change - in the given interval; and while $\Delta\mathrm{wr}[t]$ was indeed lesser than $\mathrm{r}_{bt}[t]$ before the 3.670 ms mark - it suddenly becomes greater than $\mathrm{r}_{bt}[t]$ after that mark!

Having the need to observe many variables quickly at different levels of detail with null interpolation, we immediately thought of a *multitrack* waveform interaction, in essence the one provided by audio editors like Audacity (as shown on Fig. G.23): where waveform tracks can be quickly zoomed in and out on both the time and value axis, and arbitrarily rearranged in terms of vertical track position; and where markers can be set at arbitrary positions in time, and quickly navigated to and fro, regardless of the zoom state. Unfortunately, neither `kst` nor `gnuplot` alone offer such interactive experience by default, in the versions we tried. Accidentally, we stumbled upon a relatively unadvertised library for Python, M. Tsuchida's `Xnuplot`. It is intended to provide a binary interface between Python and `gnuplot`, while allowing for the use of Python's scientific computing `numpy` package; which facilitates a rather optimized access to `gnuplot`, which works from Python 2.7. Eventually, we arrived at a GUI, which at least to an extent could simulate the multitrack interaction of audio editors like Audacity: we called this program `multitrack_plot.py`, and it is shown on the screenshots on Figure G.31 and Figure G.32 (note that these screenshots show the entire scope of plot tracks in the given case - while in typical use the window is smaller, and there is a vertical scrollbar on the right instead).

Before we address Figs. G.31 and G.32, let us look at some specifics of `multitrack_plot.py`. First of all, the acquired data from `ftdi_profiler` is formatted as a space-separated values text file, and then it is automatically parsed by the library's `numpy.genfromtxt` function into binary data; this binary data can be saved as a cache, and then loaded at subsequent runs - greatly shortening the program start-up times, as it obviates the need to re-parse the textual data. Thereafter, we have the data accessible as `numpy` arrays, which can also have named columns; and all variables that we derive thereafter, are obtained through manipulation of such arrays. Unfortunately, the issue of null-interpolating the arrays popped up again, and among our attempts, we even tried to repurpose the `interp1d` function from the `scipy` Python library - however we either encountered calculation problems (such as generation of NaN

[not-a-number] values), or the algorithms were simply too slow. Ultimately, we had to develop our own null-interpolation functions, two of which are shown on listings G.6.3:1a and G.6.3:1b.

Listing G.6.3:1a: Function getNPsaZeroCInterpolatedOver

```
def getNPsaZeroCInterpolatedOver(aa, ats, avals, bb, bts,
       bvals, ii, iis, fill_value=0.0):
  atmin, atmax = aa[ats][0], aa[ats][-1]
  btmin, btmax = bb[bts][0], bb[bts][-1]
  aail = [] ; lastvala = None
  bbil = [] ; lastvalb = None
  for itz in ii[iis]:
    a_outrange = (itz<atmin or itz>atmax)
    a_exists = itz in aa[ats]
    b_outrange = (itz<btmin or itz>btmax)
    b_exists = itz in bb[bts]
    if a_outrange:
      vala = (itz,fill_value)
    elif a_exists:
      vala = aa[ aa[ats]==itz ][[ats, avals]][-1:]
      vala = tuple( vala[0] )
    else:
      vala = (itz, lastvala[1])
    aail.append(vala)
    lastvala = vala
    if b_outrange:
      valb = (itz,fill_value)
    elif b_exists:
      valb = bb[ bb[bts]==itz ][[bts, bvals]][-1:]
      valb = tuple( valb[0] )
    else:
      valb = (itz, lastvalb[1])
    bbil.append(valb)
    lastvalb = valb
  a_npz = np.array( aail, dtype=[(iis, ii[iis].dtype), (
          avals, aa[avals].dtype)])
  b_npz = np.array( bbil, dtype=[(iis, ii[iis].dtype), (
          bvals, bb[bvals].dtype)])
  return a_npz, b_npz
```

Listing G.6.3:1b: Function getNPsaZeroDInterpolatedOver

```
def getNPsaZeroDInterpolatedOver(aa, ats, avals, bb, bts,
       bvals, ii, iis, fill_value=0.0):
  atsu,atsuind,atsuinv= np.unique(aa[ats], return_index=
          True, return_inverse=True)
  btsu,btsuind,btsuinv= np.unique(bb[bts], return_index=
          True, return_inverse=True)
  ia = ii[iis] ; av = aa[avals]; bv = bb[bvals]
  npnza = np.nonzero( np.setmember1d(ia, atsu) )[0]
  npnzb = np.nonzero( np.setmember1d(ia, btsu) )[0]
  out_of_bounds_a = np.logical_or(ia < atsu[0], ia > atsu
          [-1])
  out_of_bounds_b = np.logical_or(ia < btsu[0], ia > btsu
          [-1])
  npnzae = np.ediff1d( npnza , to_end=[1]*(len(atsu)-len(
          npnza)+1) )
  npnzbe = np.ediff1d( npnzb , to_end=[1]*(len(btsu)-len(
          npnzb)+1) )
  ainds = np.repeat( np.arange(0, len(atsu)), npnzae )
  binds = np.repeat( np.arange(0, len(btsu)), npnzbe )
  avalid = av[np.zeros(len(ia), dtype=np.intp)]
  avalid[~out_of_bounds_a] = av[atsuind][ainds]; avalid[out
          _of_bounds_a] = fill_value
  bvalid = bv[np.zeros(len(ia), dtype=np.intp)]
  bvalid[~out_of_bounds_b] = bv[btsuind][binds]; bvalid[out
          _of_bounds_b] = fill_value
  a_npz = np.zeros((len(ia),), dtype=[(iis, ia.dtype), (
          avals, av.dtype)])
  a_npz[iis] = ia; a_npz[avals] = avalid;
  b_npz = np.zeros((len(ia),), dtype=[(iis, ia.dtype), (
          bvals, bv.dtype)])
  b_npz[iis] = ia; b_npz[bvals] = bvalid;
  return a_npz, b_npz
```

The functions `getNPsaZeroCInterpolatedOver` and `getNPsaZeroDInterpolatedOver` on listings G.6.3:1a and G.6.3:1b are both called in the same manner: they accept two main input arrays (with the labels of their timestamp and value columns), and a third input array (with its timestamp column label) which needs to be calculated separately beforehand, and contains the union of the timestamps in the main input arrays; the function then returns the data in the input arrays, null-interpolated over the timestamps in the third input array, as two new arrays. The functions on both listing G.6.3:1a and G.6.3:1b return the exact same results for the same input. These functions illustrate well the issue of *vectorization* (where "vector" is again understood generically as an array of numeric values), especially relevant as an optimization technique in the Python/`numpy` environment. Namely, in a preliminary test we did, with input arrays of 4 and 13 samples (respectively) over the same range and non-repeating timestamps (i.e., the union contains 17 timestamps), the `*C*` version completes between 13 ms to 19 ms, while the `*D*` version completes in 1.4 ms to 1.6 ms - so in the worst case, the `*C*` version is some 13.5 times slower than the `*D*` version! This could be even more pronounced for larger data: for our actual data, we have experienced the `*C*` version take longer than 20 minutes, while the `*D*` version would complete in under a minute, for the same data!

The reason for this is vectorization: listing G.6.3:1a shows that the `*C*` version runs a `for` loop, with conditionals inside for *each* element of an array; the problem is that the loop is defined on the level of Python, which is a scripting (interpreted) language - therefore, to process each element of the array, the program context needs to return to Python, so it can iterate towards the next element, which costs processing cycles (and ultimately, time). On the other hand, there are no Python loops on listing G.6.3:1b for the `*D*` version: it is fully "vectorized", in that each time a Python line of code is executed, a `numpy` function is called - with *entire* arrays as arguments; thus the `numpy` function can proceed within its `C` library domain (with all the optimizations implied in that), and process the arrays in their entirety, before returning to the Python context for the next line of code. While vectorization may thus sound as a panacea, this would be a great time to remind ourselves, that - just as any other user-space code - also the `C` library implementation of `numpy` is subject to kernel pre-emption (as on Fig. G.2).

Beyond this, most of our attention in this stage was focused on deriving variables for visualization - one of which we'll describe more closely. Recall first that here, we're trying to write 64 bytes at a time, with a period of 362.8 μs; but we're also aware that the periodic kernel timer function can be preempted, in which case a write would occur later than the period time - and that in case of a preemption, the kernel can queue another write immediately after the delayed one (in quick succession); furthermore, the call to `ftdi_write()` returns immediately, without reporting on a status of the actual transmission of bytes via USB. Thus, we could say, that each time a write (also those in quick succession) returns, we are informed that the bytes are *queued* for sending (not that their sending transmission finished). If we now assume that each such queuing write, is a "fill" of a leaky bucket, which empties with the USB data rate f_B (as in Eq. G.4.5) - then we can calculate to what extent is the entire write buffer filled, each time we're registered/logged a new write; this is graphically shown on Figure G.30.

The top of Fig. G.30 shows, for reference, the time period required to send a packet of 64 bytes as per the audio rate, T_{aB} (as in eq. G.4.6; here 362 μs); for comparison, in the row below, are the packets showing the time it would take to send the same 64-byte packet as per the USB-serial rate, T_B (here 320 μs). The first two rows show the idealized periodic response; while the rows below simulate a more realistic situation: sometimes the packets are written late, and sometimes in quick succession (in which case, we show the packets below each other, for legibility). The start times of these packets are mapped to the plot below on Fig. G.30. When a packet comes in, we plot a sudden jump of 64 bytes upwards; then, the leaky bucket operation is symbolized with a downward arrow; when the downward arrow crosses the abscissa, the queued data would have been completely sent - this time is additionally visualized as forming a box around the downward arrow. For those packets that do have time to completely "leak out", like the first one on Fig. G.30, the time to "leak out" is exactly T_B. However, in case of a quick succession, the jump of 64 bytes

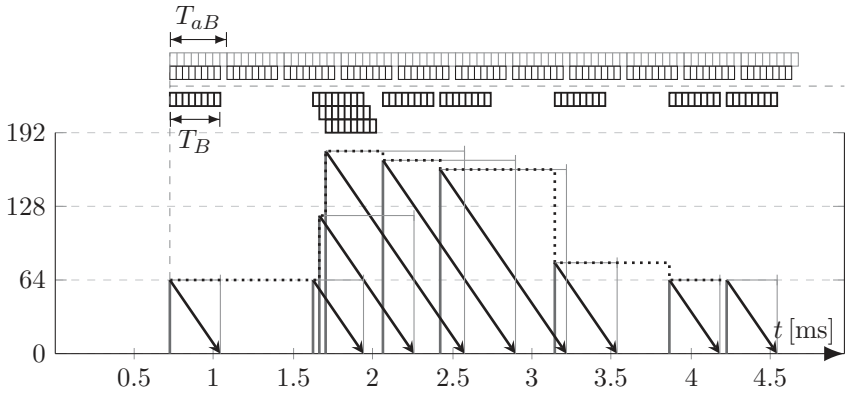


Fig. G.30: Illustration of the algorithm for estimating queued bytes for FT232 writes.

would be added to the amount "not yet leaked out", which would correspond to the height of the downward arrow at that time; as such the time to fully "leak out" increases. But even in case of quick succession - as soon as all the queued bytes have been fully transmitted, the same behavior as for the first packet is restored again. We could describe this behavior on Fig. G.30 mathematically as:

$$ftq[i] = \begin{cases} bpw, & \text{if } ftq[i-1] < \dfrac{t[i]-t[i-1]}{T_B} \cdot bpw \\ ftq[i-1] + bpw \cdot \left(1 - \dfrac{t[i]-t[i-1]}{T_B}\right), & \text{otherwise} \end{cases} \tag{G.6.1}$$

where i is the sequential order index of the packet (0, 1, 2,...); $t[i]$ is the timestamp position of packet i, bpw stands for "bytes per write" and is, in this case, 64; and ftq stands for the amount of queued bytes at time $t[i]$, that includes the contribution of the packet i. Since in actuality, we calculate eq. G.6.1 only at the timestamp for any given packet i, we would expect the variable ftq to be visualized as the dotted line on Fig. G.30 in context of null-interpolated step plot. As an example implementation of the algorithm in eq. G.6.1, below in listing G.6.3:2 is a code snippet in the Perl language, that we used to generate simulated data for Fig. G.30.

Listing G.6.3:2: Perl language implementation of eq. G.6.1

```
# $ftpd is FT232 period = T_B; @ts is array of timestamps; $bpw=64; @ftq is array
for ($ix=1;$ix<=9;$ix++) {
  $tsd[$ix] = sprintf("%.06f", $ts[$ix]-$ts[$ix-1]);
  $tsdb = sprintf("%d", ($tsd[$ix]/$ftpd)*$bpw);
  $ftqh = $ftq[$ix-1]-$tsdb;                                          # helper 1
  $ftqhb = $bpw-$tsdb;                                                # helper 2
  $ftqhh = ($ftqh<0) ? 0+$bpw : ( ($ftqhb<0) ? $ftqh+$bpw : $ftq[$ix-1]+$ftqhb ) ; # helper 3
  $ftq[$ix] = $ftqhh ;
  $gstr .= "$ts[$ix]␣$ftq[$ix]\n";                                    # add string entry
}
```

Now, let us take a closer look at the visualization of data, acquired by `ftdi_profiler`, in the `multitrack_plot.py` program - as shown on the screenshots on Figs. G.31 and G.32. Fig. G.31 shows a visualization of a 30-second `ftdi_profiler` acquisition, using 9 tracks (with multiple plots inside); overruns are shown as red vertical lines spanning all tracks. The `multitrack_plot.py` GUI draws a vertical cursor at the mouse location across all tracks, allowing for easy selection of a region to zoom into; there is also limited keyboard interaction for navigating the zoom history. Fig. G.32 shows about 30 ms of a zoomed-in region around the second overrun (at around 7.97 s) on Fig. G.31; the ranges of the respective track plot ordinates are auto-adjusted to the values shown. A brief rundown of the variables plotted is given on table G.1.

Figures G.31 and G.32 show some of our intent to predict an overrun. For instance, track 4 shows the `atso2thr` variable; this is simply a threshold, which at start we've set to various values (on the figures, that value is 256), and then each time an overrun is detected, we increase that threshold by the "bytes per write" amount (64). We hoped that some of the variables would cross this threshold right before an overrun in specific manner, however that is difficult to determine: Fig. G.32 shows that `wrdltz` crosses the threshold all the time; while `wrdlt2cs` seems to cross it right before the overrun happens. However, our hope in the write queue algorithm (as per Eq. G.6.1 and Fig. G.30) as a potential overrun detector, seems to have been misplaced: note that on track 7, Fig. G.32, the `wftq1` variable is more or less constant both before and after the overrun - and as such tells us nothing about a potential overrun condition. We found that the behavior of the read function, where there is a quick succession of reads (like in a loop), followed by a longer wait time that can be seen as a period (as visible on Fig. G.30, or the variables `rlen2` on track 6, or `rtoe2` on track 9 on Fig. G.32), made it difficult for us to perceive the overall behavior of the read side – especially graphical estimation of the total bytes transferred during read in a single long period. This is why we've implemented a (vectorized) piecewise cumulative sum algorithm in `numpy`, which sets a time threshold - and for all successive reads that come closer than this threshold, their values are summed; and finally the first item in the quick succession is updated with this cumulative sum, while the rest of the items in the quick succession sequence are dropped, resulting with a filtered version of the variable (we typically add "cs" in the names of such variables). This allows us to view the read operation as roughly periodic at about 1 ms, which is visible on the variables `rlen2cs` on track 7, or `ts2csdl` on track 8, Fig. G.32.

Here, we should note that the overrun zoomed on Fig. G.32, was signaled by line status byte 0x02 (which we understood earlier to mean an overrun in the FIFO RX buffer of the FT232) in that particular acquisition. In general, the plot on Fig. G.32 seems to indicate that - for that particular overrun - the problem is not so much in the write procedure, but in the read one: the read rate in bytes per second (`rbps2cs`, track 2) experiences a sharp drop right before the overrun is detected; `ts2csdl` (track 8) shows that instead of the usual 1 ms - right before the overrun, a read function did executed late: approx. 2 ms

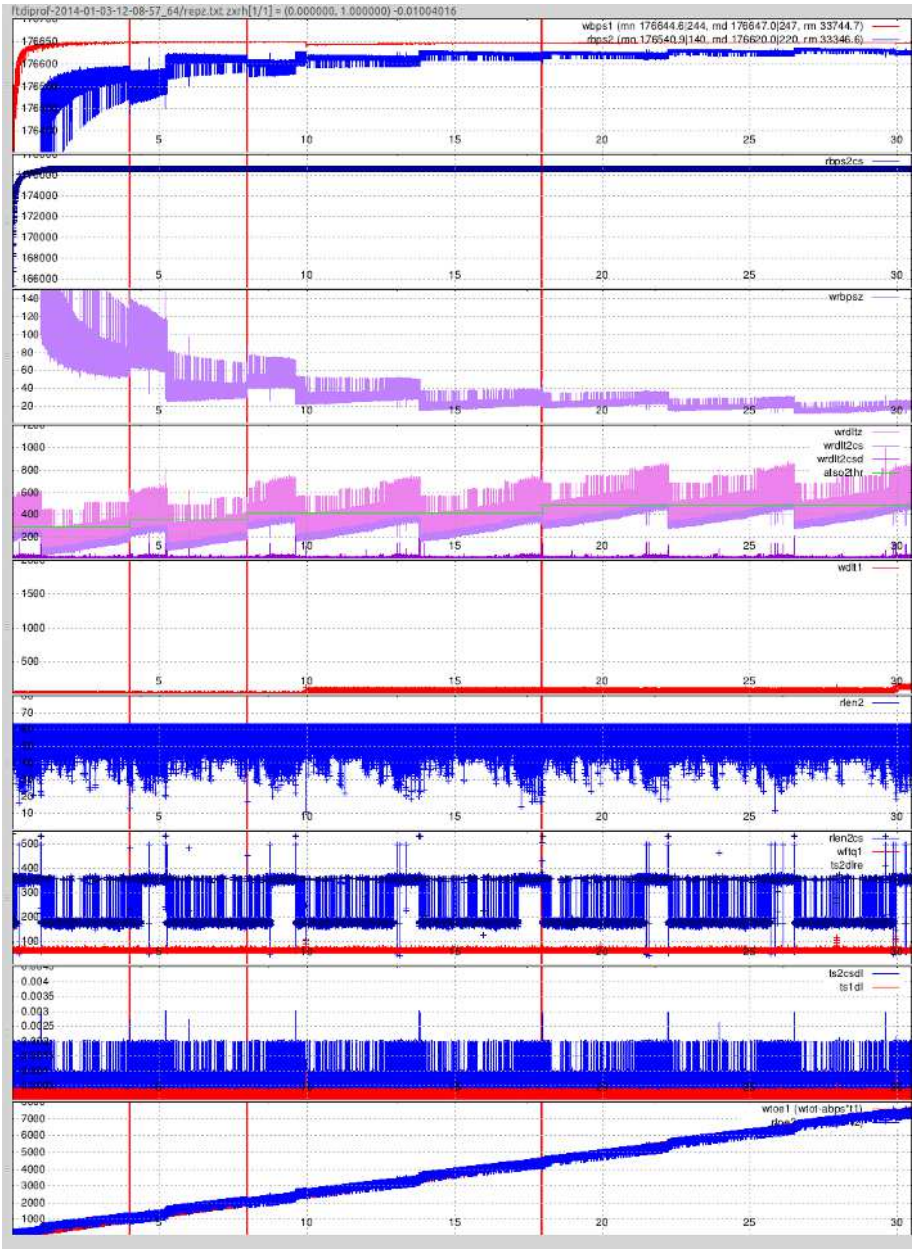


Fig. G.31: `multitrack_plot.py` showing a 30-second long acquisition

Table G.1: Overview of variables shown (per track) on Figures G.31, G.32

Track	Variables	Notes
1	`wbps1`, `rbps2`	write and read bytes per second (total / expired time); in respect to their own timestamps (1: write, 2: read)
2	`rbps2cs`	read bytes per second "cumulative sum": reads in quick succession (apart below a threshold) are summed together, and that value is used at the first read in the quick sequence (so the rest of the sequence does not have to be plotted; i.e. is filtered)
3	`wrbpsz`	the delta (difference) wbps-rbps; in respect to the union of both timestamps (z)
4	`wrdltz`, `wrdlt2cs`, `wrdlt2csd`, `atso2thr`	the delta (difference) between total write and read bytes (w_{bt}-r_{bt}), expressed through union of timestamps (z); read timestamps with "cumulative sum" filtering of quick successions (2cs), first differential of previous (2csd); and an attempt to find a threshold level, which when crossed would indicate an overrun, but also increases for "bytes per write" upon an overrun (2thr) - all expressed in read timestamps (2)
5	`wdlt1`	difference (in bytes) between how much the written bytes total should be (based on elapsed time), and the actual total of written bytes; in respect to own (1) timestamps
6	`rlen2`	size (length) in bytes of the last read; in respect to own (2) timestamps
7	`rlen2cs`, `wftq1`, `ts2dlre`	length in bytes of the last read "cumulative sum" - quick successions filtered (2cs), queued write bytes (ftq1, eq. G.6.1), expected amount of bytes to be read based on the time delta from last "cumulative sum" read (2dlre); in respect to own (1: write, 2: read) timestamps
8	`ts2csdl`, `ts1dl`	time delta from last "cumulative sum" read (2csdl), time delta from last write (1dl); in respect to own timestamps
9	`wtoe1`, `rtoe2`	total write (toe1) and read (toe2) error: difference between the expected total of read/written bytes as per elapsed time and the audio rate, and the actual total of read/written bytes; in respect to own timestamps

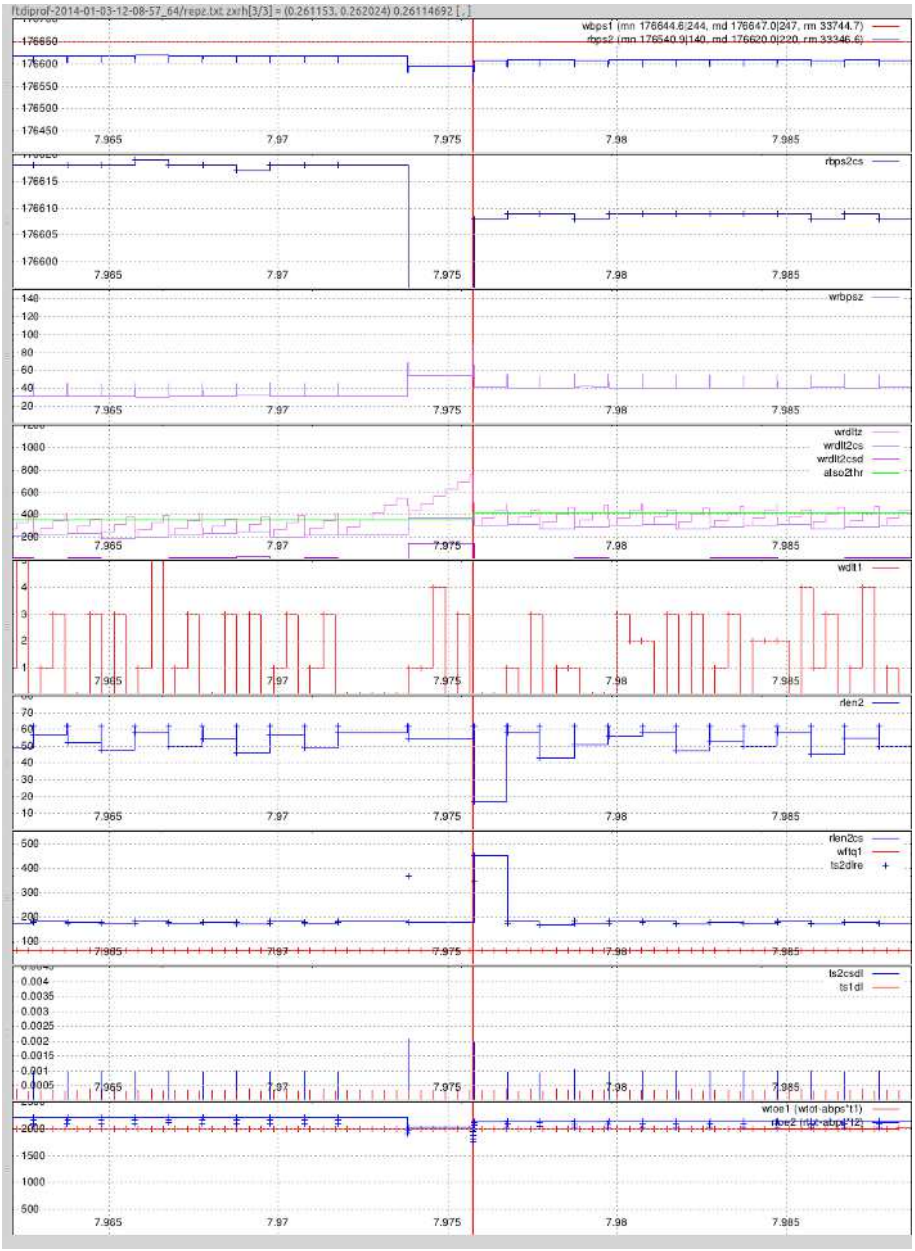


Fig. G.32: Same acquisition as Fig. G.31, zoomed in around an overrun

after the previous one. Finally, at this moment, `ts2dlre` (track 7), which is the expected amount of bytes to be read based on elapsed time, is greater than `rlen2cs`, the actual amount delivered (track 8) - meaning that this instance of the read function is not only late, but it also didn't compensate for that lateness by contributing correspondingly increased amount of data, either; and during the entire time this happens, the writes continue unabashed. Had this been an overrun predictor condition, we could have implemented the algorithm in the read function of our kernel driver; once it detected that reads are coming in delayed with not enough data, it could have signaled (say, through a mutex variable) to the periodic timer function doing the writes, that it should bail out early and stop writing until further notice. Note that even if this fixed the FTDI overrun errors, it could have also re-introduced detection of timing errors in ALSA, or a related audio library, at the same time! However, the conditions identified here, while they seem reasonably related to an overrun, are not absolute predictors of one - mainly because they (i.e. delayed reads where `ts2dlre` > `rlen2cs`) occur many times in a 30-second acquisition, *without* triggering an overrun; and so there are definitely more of them, than the number of detected overruns. It is possible such a delayed read could be a necessary, but not a sufficient condition for a FIFO RX overrun - which in turn means, there must be something else at play that finally causes the overrun, which we haven't identified yet.

There were factors that pointed to, that we wouldn't gain much new insight with `multitrack_plot.py`: in spite of the timestamp which is logged when the timer function runs, we're fundamentally uncertain about the time, when the bytes written are actually transmitted over USB; information from `usbmon` could have helped, but it's timestamps being in different domain introduce a further level of uncertainty; thus we'd have difficulties in actually predicting the state of the FIFO RX buffer of the FT232, based solely on PC-side information. Had we been able to read the state of the FIFO RX, instead of predicting it, it may have been possible to use it for traffic shaping purposes. We should also note, that in particular `multitrack_plot.py` currently suffers from "research code" quality: for instance, it is actually used by loading it in other Python scripts, but it is not written as a library (as software development practices would otherwise recommend, for that context of use). As with the rest of this article, all of the software developed by us, discussed in this section, is released as open-source, and available online via [1].

G.7 Debugging facilities - overview

From the discussion so far, we could observe that in the debugging approaches we have touched upon, three distinct phases become apparent: obtaining data, parsing data, and visualizing data. Parsing data, in this context, usually involves processing plain-text log files, with the end result being a delimiter (space or comma) separated value file (again in plain-text format). For the parsing

stage, we have used a variety of scripting languages, such as `awk`, Python or Perl - which is a standard practice for those kinds of tasks. In the discussion so far, we have also addressed several applications that we used for the visualization of data - such as `gnuplot`, `python-matplotlib`, `python-chaco` and `kst`; while there is also other open-source Linux software that could be applicable, mainly GUI waveform viewers (such as `gtkwave`), the need for customized plotting of numeric values as waveforms, usually made us turn to script-controlled plotting engines (mainly `gnuplot`).

In terms of obtaining debug data, however, it was not always clear to us what facilities exist for our development platform; and which of those would be applicable to our context of use - which is the operation of the ALSA audio subsystem, and related hardware subsystems (e.g. USB), on the kernel level. Because of that, in the rest of this section, we provide a brief overview, in no particular order, of tools and approaches we have met (and in some cases, used) during the development outlined in this article; and their applicability for our development context and platform. Since super-user (`su`, or `sudo`) permissions are required to load a kernel module in the Linux kernel - all of these approaches will, at least partially, also require super-user permissions.

- `printk()`: This function is the main Linux kernel API facility, allowing for timestamped output / logging of arbitrary data; and as such, is widely used. Has to be manually inserted in kernel code; requires recompilation. The output is found by reading the `/var/log/syslog` file, or by running `dmesg`. There are facilities that allow the `syslog` to be broadcasted on the local network and captured on a different computer, which is extremely useful in case of a kernel module bug, which results with a kernel "Oops" or a freeze (which typically require a hard restart, that the `syslog` contents do not otherwise survive).

- `trace_printk()` / (`ftrace`): This function, which is a pendant to `printk()`, is a part of the `ftrace` debug facility of the Linux kernel (included since version 2.6.27). As such, it writes into the `ftrace` buffer, and thus has less overhead than `printk()`. The output is found by reading the `/sys/kernel/debug/tracing/trace` (or related) file. One of the more important facilities of `ftrace` for us is the "function graph", which is a timestamped and nested log that records entries and exits of all kernel functions, that had ran on the PC during a debugging trace acquisition session.

- `gdb`: The GNU Debugger is a command-line application, whose use for debugging user-space code (libraries or executables) is standard practice. It allows for setting breakpoints in code, as well as keyboard interaction to step through code: from lines of source code, down to machine instructions; and many other facilities (e.g. stack traces). It can *not* be used to debug a live kernel module in the same running instance of the OS kernel. However, there is an engine called `kgdb`, which allows `gdb` on one

computer, to control and inspect a debug kernel on a different computer (or in a virtual machine) over a serial (or network) connection – but the requirement to use a debug kernel slows it down significantly, influencing the appearance of timing errors in ALSA.

- `trace-cmd` and `kernelshark`: `trace-cmd` is a command-line application, and `kernelshark` its GUI counterpart, which primarily offer an interface to the `ftrace` debug facility of the Linux kernel. As such, apart from a slightly easier setup of `ftrace`, `trace-cmd` can also obtain the contents of the `ftrace` ring buffer as a binary file - and decode its contents (including the outputs of `trace_printk()`) into plain text output at a later time. These binary files can be seamlessly opened, and visualized, in `kernelshark` (which, however, tended to be somewhat slow on our development platform).

- `perf`: This is a command-line tool, with functionality partially integrated in the Linux kernel; originally called "Performance Counters for Linux". It can utilize hardware based counters, to keep track of how many times particular functions have executed (in both kernel- and user-space). As such, it can reveal statistical information related to performance (e.g. in which function does a program spend most of its time) - but is not meant to provide timestamp ordered data, or to reveal the order of execution of functions at a particular time.

- `OProfile`: This is a suite consisting of command-line tools, a kernel module and a user- space service. It's purpose is mostly the same as `perf`: that is, counting of events related to performance statistics. In addition, it can derive call graphs, which can show a tree of which functions are called from a given function, and how much time is spent in them – but again, it is not meant to reveal the timestamped order of execution of functions.

- `pytimechart`: This is a Python GUI application, implemented using the already mentioned `chaco` library, which is an alternative to `kernelshark` for visualizing `ftrace`, `trace-cmd` and `perf` log acquisitions. While it is intended to be a faster tool than `kernelshark`, it was problematic to install it on our development platform - and in that context, we experienced it performing somewhat slow as well.

- `kprobes`: This is an older Linux kernel API, originally contributed by IBM, which offers kernel functions like `register_probe()` for kernel-space probes, or `register_kprobe_user()` for user-space probes. It basically allows building of custom kernel modules - which, when loaded, can hook into any kernel function (even those residing in other kernel modules), and can run custom code before and after the traced function executes. It seems to be in use (or at least has been in use, historically) by `systemtap`.

- `systemtap`: This is a framework with command line tools, quite applicable to our context of use: it allows for writing instrumentation scripts - where one can define the startup of a user-space program; as well as "probe points", which run each time a particular user- or kernel-space function runs, and can print various information, including timestamps. Such a script is compiled transparently into a kernel module when ran by the command-line tool; thus, in principle it can be used to trace another kernel module without recompiling it. However, the `systemtap` script's utility depends on the existence of debug symbols in the traced modules - and therefore, sometimes recompilation of the target kernel modules is necessary.

- `dtrace`: First, let's note an ambiguity here: in the Ubuntu packages (also for our development platforms), a `dtrace` executable can be found, which is in fact just a `systemtap` script - this is not what we are addressing here. Here we refer to Sun Microsystems' DTrace (capitalized) framework, primarily intended for the Solaris OS – however with source code available for compilation on Linux, under which the main executable again has the (lowercase) name `dtrace`. It works in a similar way to `systemtap` - via instrumentation scripts that are compiled into a kernel module, and can trace other kernel modules. We managed to build the Linux version on our development platform, and even run it with some basic examples - however, what we needed were what it calls "fbt" (Function Boundary Tracing) probes, whose use unfortunately caused our OS kernel to freeze; not being able to solve this problem, we couldn't make further use of `dtrace`. Note there may be issues in compatibility of the DTrace (CDDL) and Linux kernel (GPL) open-source licenses (see comments in [55]).

- `valgrind` and `kcachegrind`: `valgrind` is a command-line application with several separate tools; the most interesting tool for us was `callgrind`, which produces call graph information (similar to the one mentioned for `OProfile`). There seem to be several GUI applications for `valgrind` generated data, among them `kcachegrind` which can visualize `callgrind` call graphs (note the name similarity to `cachegrind`, which is another `valgrind` tool, whose output can also be visualized by `kcachegrind`). While `valgrind` is often cited in context of debugging, it is important to note that `valgrind` tools cannot inspect/profile kernel-space code (at least on our development OS versions); therefore, they are usable only in a user-space context.

- `lttng` and `lttv-gui`: Linux Trace Toolkit New Generation (which supersedes the earlier `ltt`) is a software collection and a command line tool, allowing for tracing kernel- and instrumented user-space code; it includes the Linux Trace Toolkit Viewer GUI. It allows for acquisition of timestamped events, like entries and exits into functions (similar to `ftrace`'s "function graph"); however, it achieves that by loading its own kernel mod-

ules in the system. We had to build `lttng` from source to have it run on our development OS; and while the resulting software itself was usable - at the time we were ultimately interested in tracing user-space programs without recompiling/changing them; and this could not be done with `lttng`, as it requires user-space code to be instrumented, by inserting additional calls to the `lttng` API in it.

- `Perfkit`: C. Hergert's "performance recording toolkit" (not to be confused with NVIDIA's PerfKit software suite) is a GUI application, which aims, via plugins, to provide real-time visualization of `valgrind`, `ftrace`, or `perf` data. As we couldn't find much documentation on customizing the produced plots, and we otherwise weren't in need of the real-time capabilities, we didn't make use of this application during our development.

- `usbmon`: This is a built-in debug facility in the Linux kernel, allowing for acquisition of data related to USB requests and responses on the kernel level. Similar to `ftrace`, the generated data can be plain-text formatted and relatively human-readable; it can be obtained by reading system files like `/sys/kernel/debug/usb/usbmon/2u` (where e.g. the 2 refers to the USB bus number) during the USB operation. It also provides a `C` language API for collecting the data in binary format. Unfortunately, on our development OS, it is not integrated with `ftrace`, in the sense that there is no easy way to make `usbmon` events show in the `ftrace` log with a minimum of latency; and also, `usbmon` supplies its timestamps in a different format from `ftrace`.

- `tshark` and **Wireshark**: The command-line `tshark`, and its GUI counterpart Wireshark, are already established applications for inspecting network traffic; they are relevant here, because they can interface directly with `usbmon` on Linux, and capture all of the USB traffic (including the entire data payloads of USB request/response packets), in a binary file - which can thereafter be decoded, and converted to a text format, by these same tools.

- `vusb-analyzer`: This is a GUI application, implemented in Python and GTK+, released as open-source by VMware. It can visualize `usbmon` data, as well as few other types of logs. It is especially convenient, because instead of visualizing USB requests and responses separately (as we did on Fig. G.22), it takes them as the start and the end of a USB transaction - which is then visualized as a box of a limited duration in the timeline of the GUI, that also features a separate track for each USB endpoint.

- Debugging **ALSA**: To debug ALSA, one can build both the ALSA kernel modules (in the `alsa-driver` package), and the ALSA shared object library (in `alsa-lib`), in debug mode. There are several debug options for `alsa-driver` that can be enabled at compilation time; among them `CONFIG_SND_PCM_XRUN_DEBUG`, with which each substream of each sound

card will get a file, e.g. `/proc/asound/card0/pcm0p/xrun_debug` for the card 0, device 0, playback substream. Then, a number corresponding to a bitmask [56] can be echoed into this file, which will cause additional debug information to appear in the system log `/var/log/syslog`. As mentioned in Sect. G.5.1, if we want to use the debug ALSA kernel modules, the vanilla ALSA modules for the OS must be blacklisted from loading at boot - else it is impossible to remove them fully from a live system (to allow the debug versions to load). When `alsa-lib` is built in debug mode, the resulting shared object library, `libasound.so`, can be loaded separately for each call to an executable, using the `LD_PRELOAD` environment variable in a Linux shell; when it is loaded, setting an additional environment variable `LIBASOUND_DEBUG` to a number (verbosity level, see the file `alsa-lib/NOTES`) will cause additional debug messages to be printed - however, notably, not in the system log, but in the standard output stream of the respective executable that uses the library.

- Debugging **PortAudio**: The shared library `libportaudio.so` can also be built in debug mode, and loaded individually (per executable call) through the environment variable `LD_PRELOAD`. If the debug library is loaded when an executable utilizes it, it will output additional debug messages - again, into the standard output stream of the running executable.

All of the findings noted above, should hold for the stated versions of our development OSs. Note that the debug version of both `libasound.so` and `libportaudio.so` shared libraries can be inspected with `gdb`. Note also that some of these applications advertise simultaneous inspection of both user- and kernel-space code; for most of them, this is a relatively new development, afforded by a functionality in the Linux kernel known as `UPROBES`, first introduced in the 3.5 kernel series – and as such, not available on our development OSs (which forced us to use a more complicated approach with hardware breakpoints to obtain listing G.2:2b on Fig. G.2). Each of the options above offers a different trade-off between ease of setup, ease of use, detail of debug data, and influence on the timing as observed by audio subsystems; and as such, may be appropriate in different contexts. For instance, in earlier work we have successfully used a `kgdb` setup with a PC and a virtual machine, to debug a null-pointer error in an ALSA driver of ours, which otherwise caused a kernel freeze with no specific error messages; that same setup would have so much overhead and influence on timing errors, it would make it impossible to inspect the operation of running audio – however, for that purpose, an `ftrace` based setup can be used instead, as we have done earlier in this article. We do not claim this to be a complete list of debugging approaches for audio on Linux; merely a list of those approaches that we had considered, or used, throughout the development of this project.

G.8 A note on obsolescence

A major source of irritation and frustration during the development of this project, was the fact that nearly all the technology that we use in it is *already obsolete*. We were continuously faced with the dilemma to either upgrade to latest versions of software, along with the risk of discovering new bugs - or stay with the current versions, and bugs that may already have been fixed, so that this work would have a continuity with our previous soundcard related work, at least in terms of kernel series; as this article shows, ultimately we opted for the latter.

The issue of obsolescence does not refer to the software only, but also the hardware: the Arduino Duemilanove, with an FTDI FT232 USB chip, has long been obsolete, i.e. it is not produced anymore. Since we own very few devices of this kind, the possibility that the board might break in mid-project, leaving us without means to complete our research, was a constant source of pressure. The open-source nature of the Arduino may have alleviated even a condition like that, as the schematics are readily available, and it seems the PCB layout consists of only two layers, making it reconstructible even in a hobbyist lab. However, that reconstruction would still depend on availability of the exact same parts, and as such, would have introduced further uncertainties in our conclusions. The Duemilanove has long been superseded by Arduino Uno, which has the same microcontroller (ATmega328) – but instead of an FT232, it uses another Atmel microcontroller, ATmega16U2 (earlier ATmega8U2), which is programmed as a USB-serial converter (and in principle can be programmed to exhibit other USB behaviors as well). Given that there is open source code for the USB stack of the ATmega8U2/16U2 through the LUFA (Lightweight USB Framework for AVRs) project, these devices can be considered an open version of the FT232. This development certainly aligns with our goals in this project, especially after our realization of the limits set by NDAs on open development for FTDI chips.The issue of hardware obsolescence can be said to have touched our development PCs, as well: the NetColors brand of netbooks, one of which we originally procured from China, does not seem to exist anymore (along with the website we ordered it from).

Furthermore, the Linux kernel 2.6.* release series, that we developed the software in this project for, has likewise been long obsolete. As of July 2014, the current mainline kernel is at version 3.16-rc4, and the stable kernel at version 3.15.5. The operating systems we used that were based upon the 2.6.* kernel series, Ubuntu 10.04 (Lucid) and 11.04 (Natty), are likewise long superseded; as of July 2014, the current Ubuntu OS, which is also a Long Time Support (LTS) version, is version 14.04 (Trusty), which is based on Linux kernel 3.13. These changes influenced our development as well, in that some of the earlier mentioned tools we used, which we had to build from source, failed to build at their latest version on our development OSs; so we had to check out earlier revisions to make them work. However, if there is one benefit from our staying at the 2.6.* series, it is the relatively wide availability of literature (like [8], [7])

that addresses precisely the 2.6.* kernel series, and the changes introduced in it – which should help make our work easier to contextualize in educational terms, even if this kernel series is obsolete in practical terms.

Even further issues of compatibility emerge from relatively unrelated development of different OS components and user-space software. Maybe the most major transition, relevant to us, is the transition of the Gnome desktop environment, which we utilized in our development OSs, from version 2 to version 3. The differences can be drastic: for instance, unlike Gnome 2, Gnome 3 requires hardware acceleration on graphic cards to start, and provides a fallback mode for hardware that does not meet its requirements. On the other hand, the codebase of Gnome 2 has been forked, and now exists as the MATE desktop environment. The Gnome desktop environment uses the GTK+ (originally GIMP Toolkit, not Gnome Toolkit) library to build its GUI widgets, and thus the transition also encompasses the underlying GTK+ versions. Most importantly, the APIs have changed as well, and there is no API compatibility between GTK+ 2 and 3 – as we'll see shortly, this change was more significant for us, in the role of developers using the GTK+ library.

Yet another transition that influenced our development, was the transition of the Python scripting language from version 2.7, which was the default on our development OSs, to version 3. This transition introduces deep changes to the language which are not compatible between versions; as one example, the basic statement `print "test"`, completely valid in 2.7, will raise a syntax error in Python 3, which halts the execution of the script - there the correct syntax is required to be `print("test")`. This impacted our development, because among the reasons for our development and release of open source software, was the expectation that once released, our software will be available for use, without further maintenance from our side. What we didn't clearly realize until this transition, is the fact that while our software may remain *available* (say, for download) in the future, that does not mean it will remain *viable* for use: its usability will be limited by compatibility with the interpreter engines, shipped with ever newer versions of OS distributions.

Having realized this during the development of `numStepCsvLogVis.py` (Section G.4.1), we had to put additional effort to make the same script run under both Python 2.7 and 3.2. Part of this was identifying `Tkinter`, the default GUI library for Python, as one that did not suffer such drastic changes - which allowed us to provide a GUI that runs under both Python 2.7 and 3.2 for the same, single script `numStepCsvLogVis.py` file. In addition to this, `numStepCsvLogVis.py` also handles some of the changes between the respective `numpy` and `matplotlib` library versions related to Python 2.7 and 3.2, respectively; these efforts will hopefully afford a slightly longer usable lifetime of this software. For `multitrack_plot.py` (Sect. G.6.3), we didn't have such luck: for one, it was programmed somewhat in haste, and additionally, we needed an easy access to a GUI element that would behave like a horizontal track in a multitrack setting - and the `pygtk` GUI library for Python 2.7, as described earlier, affords this. Only later did we realize that thus we've made this program

somewhat unportable: even if we can replicate the syntax constructs, that allowed us to run the same general Python code in both 2.7 and 3.2 earlier - the underlying GTK library, as we mentioned, is not compatible between its own versions 2 and 3. In fact, in turns out that `pygtk` refers strictly to the API interface to GTK+ 2, when used through a call like `import pygtk`; Python 3 by default uses the API interface to GTK+ 3, referred to as PyGObject, which is used through a call like `from gi.repository import Gtk`. Needless to say, the syntax and functionality provided in `gi.repository.Gtk`, is not compatible with the corresponding facilities in `pygtk` – and this makes it difficult for us to develop a version of the `multitrack_plot.py` application that can run under both environments.

To summarize: the domain of free/libre/open-source software and hardware, is by no means immune to the phenomenon of obsolescence, which we can perceive as the byproduct of the development of technology generally. While, in general, one has to live with the notion that such changes in development are for the better - obsolescence can also exert a pressure, especially on developers of research code like us: whose goals are often to develop and release demonstration software, limited in scope, and without the intent for continuous maintenance of the same. This is likely based in a certain underlying promise, that digital technology in general, will ultimately allow for something, best expressed in the old Java slogan: "write once, run anywhere" – our experience shows that, while quite powerful, this notion may be better characterized as hope, rather than as promise. Still, FLOSS software affords one important freedom: the freedom to run obsolete versions of OS and user-space software, if the task at hand demands it; however, even this freedom is only as relevant, as the machines capable of running the code in question are readily available.

G.9 Conclusion

In this article, we provided an overview of our development efforts, to extend our previous soundcard work for ALSA, to CD-quality full-duplex operation. We have outlined the successful solution to the issue of a virtual ALSA driver operating at those settings; while in terms of hardware, we couldn't provide a complete solution for a full-duplex, CD-quality AudioArduino driver. However, we believe we may have identified, at least partially, the source of the problem in the FT232 buffer overruns - and the limits to meaningful development in an open-source context that might address it. Furthermore, in outlining our development approach, we have provided discussions on relevant background problems, such as: the issue of kernel preemption, the difference between the standard and high-resolution Linux kernel timer API, and the effect that has caused proper operation of our previous, low-speed drivers, even in the presence of jitter – as well as the relationship of DMA operation to ALSA, and details about USB traffic on the kernel level. Such details do not necessarily form the core of professional inquiry even in strongly related fields, such as electronic

music instrument development.

Even as part of the electronic music instrument community, we found it hard to track down material that addresses practical problems in digital audio development, especially ones expressed in terms of free/libre/open-source tools, which we could have used as a basis; and the topics we discussed here haven't otherwise formed a significant part of our professional culture previously. In other words: had we previously found discussions of problems, inherent in the transition of PC digital audio to CD-quality, and expressed through free software examples and tools - we would not have felt compelled to spend nearly a year and a half on research & development leading to the production of this article and supporting materials. On the other hand, the topics discussed herein are, in themselves, nothing special - and there certainly exist fields of computer science, where these issues have long formed the core of the curriculum. However, if those topics are not part of one's professional background, one is bound to spend massive amounts of time on it - and this is where, we believe, lay the greatest contribution of this article: for all those, especially hobbyists and newcomers to the electronic music instrument community, for whom these topics do not necessarily form a part of their professional background - this article and supporting material offer the opportunity, especially through that part of our examples that can run on a PC independently of external hardware (and can in principle be carried out "at home"), to shorten the time spent on gaining an insight into full-duplex, CD-quality digital audio, to a fraction of the time that *we* spent (on achieving the same).

This can also be seen from an educational perspective, especially in light of our decision to keep this project expressed in terms of the same OS / kernel series, as our previous work: with this, we offer educators a complete package, describing issues in digital audio development from both the software and the hardware side, and spanning audio settings from simplex, single channel, 8-bit, 8 kHz, to full-duplex, dual channel (stereo), 16-bit, 44.1 kHz - all expressed in terms of a single/unified operating system and environment. This should allow educators, who are considering implementation of practical exercises relating to digital audio, insight into expected problems and possible solutions with free/libre/open-source tools - should they find it applicable to port the kind of examples described in our work, to their particular hardware and software platforms.

As a side note, for all our relations to electronic music instruments, the development described in this article was surprisingly silent; in fact, we cannot recall a single instance, where we actually heard audible sound - or even felt that a test with actual audio files would be warranted. And, somewhat ironically, the development process also showed that one of the major problems in understanding PC digital audio on this level is, in fact - visualization. We mentioned previously, that one of the major problems that we experienced, was that not just new types of visualization, but sometimes also incremental changes to existing types, can require a level of effort equivalent to starting a new software project, with the corresponding cognitive strain - which can often

lead to fatigue. In response, we would like to propose a concept for a GUI application as a front end to various debugging tools, which we believe would alleviate some of these issues.

Essentially, this application would simply relate the most crucial elements in this kind of development: userspace & kernel space source code; scripts for source code build processes; scripts for running executables and acquisition of their debug output; and scripts for visualization, animation or sonification of the debug output data. So, the application would consist of a single window, with multiple tabs, each corresponding to one of the major phases in data visualization. In the first tab, source files for both kernel- and user-space software are specified (through a list, or a regular expression); as well as a script (or a program) which would be able to run the resulting software, possibly multiple times and with variable parameters for each test run, and obtain debug logs from the runs (possibly consisting of multiple files per run), whose naming is also set in this tab. In the second tab, a list (or a regular expression) selects valid input log files (obtained through test runs in the first tab), and a script (or program) is set, which will parse the input log files, and generate text-formatted, delimiter-separated-values data files, whose naming policy is also set in this tab. In the third tab, a list (or a regular expression) selects valid input data files (obtained through the parsing procedure in the second tab), and a script is set, which determines how the data is to be plotted; a possible output from this tab could be a binary file, containing the plot data. A fourth tab would contain the plot graphics GUI, where the binary files from the previous tab would quickly loaded and visualized: the desired attributes here would be: quick zooming in and out of ranges and multitrack operation (like in Audacity), but also a possibility for horizontal or vertical arrangement of multiple plot tracks, possibility to move plots from one track to another and displace them in either axis direction, possibility for quick transparent overlap between displaced tracks, and possibility to set markers, manually or according to an algorithm, and to perform measurements on them; and finally, to allow for export bitmaps of the plots based on variable parameters, that could be used for producing animations. The settings in these four tabs would constitute a project; the important thing about this application would be to remain language agnostic (i.e. the user would use their language of choice to implement parsing or plotting scripts), focusing only on setting up relationships between input sources, data, and plot files; and to allow for incremental changes to files, listed as scripts or source files (but not log or data files) within a project, to be saved in an underlying revision control system (possibly `git`). This kind of application would address one of the problems we recognized during this type of work, which is cognitive strain, for example due to the proliferation of names of debugged variables (and keeping track of their role in plots) – and it could allow for easier development of test cases, especially for software bug submissions.

In conclusion, we have provided an overview and analysis of issues, both in hardware and in software, involved in the transition to CD-quality digital

audio on the PC - by using an entirely free and open source toolchain (with the exception of the Logic analyzer software, for which we've identified an open-source alternative). We believe this will be of use, as practical introductory material into high-fidelity, multi-channel digital audio topics, primarily for the hobbyist and electronic music instruments community – but, possibly, also for the wider academic community, as potential laboratory exercises in the general area of digital signal processing. Even if the technology used in our work is already obsoleted in terms of state of art - we hope that, by aiming to perceive the devices and programs used as generically as possible, this article (along with the rest of our soundcard related series) and related materials (available via [1]) will still be useful - if only as a conceptual guide, or map, to expected problems on this level of development. Ultimately, we hope that this project contributes to a possible basis, that would support the further proliferation of open soundcard and audio hardware projects - implemented in ever newer hardware and software platforms.

References

[1a] Smilen Dimitrov, "Extending the soundcard for use with generic DC sensors", in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2010)*, Sydney, Australia, Jun. 2010, pp. 303–308, ISSN: 2220-4792, ISBN: 978-0-646-53482-4. URL: http://imi.aau.dk/~sd/phd/index.php?title=ExtendingISASoundcard.

[2a] Smilen Dimitrov and Stefania Serafin, "Minivosc - a minimal virtual oscillator driver for ALSA (Advanced Linux Sound Architecture)", in *Proceedings of the Linux Audio Conference (LAC 2012)*, Stanford, California, USA, Apr. 2012, pp. 175–182, ISBN: 978-1-105-62546-6. URL: http://imi.aau.dk/~sd/phd/index.php?title=Minivosc.

[3a] ——, "Audio Arduino - an ALSA (Advanced Linux Sound Architecture) audio driver for FTDI-based Arduinos", in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME 2011)*, Oslo, Norway, May 2011, pp. 211–216, ISSN: 2220-4792, ISBN: 978-82-991841-7-5. URL: http://imi.aau.dk/~sd/phd/index.php?title=AudioArduino.

[4a] ——, "An analog I/O interface board for Audio Arduino open soundcard system", in *Proceedings of the 8th Sound and Music Computing Conference (SMC 2011)*, Padova, Italy: Padova University Press, Jul. 2011, pp. 290–297, ISBN: 978-8-897-38503-5. URL: http://imi.aau.dk/~sd/phd/index.php?title=AudioArduino-AnalogBoard.

[5a] ——, "Towards an open sound card — a bare-bones FPGA board in context of PC-based digital audio", in *Proceedings of Audio Mostly 2011 - 6th Conference on Interaction with Sound*, Coimbra, Portugal, Sep. 2011, pp. 47–54, ISBN: 978-1-4503-1081-9. DOI: 10.1145/2095667.2095674. URL: http://imi.aau.dk/~sd/phd/index.php?title=AudioBareBonesFPGA.

[6a] ——, "Open soundcard as a platform for practical, laboratory study of digital audio: a proposal", *International Journal of Innovation and Learning*, vol. 15, no. 1, pp. 1–27, Jan. 2014, ISSN: 1471-8197. DOI: 10.1504/IJIL.2014.058865.

[1] Smilen Dimitrov, "Comparing CD-quality soundcard drivers homepage", web page, 2014. URL: http://imi.aau.dk/~sd/phd/index.php?title=ScdComparison (visited on 07/15/2014).

[2] Chip Chapin, "CD-DA (Digital Audio) 1", Apr. 16, 2005. URL: http://www.chipchapin.com/CDMedia/cdda1.php3 (visited on 02/24/2015).

[3] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978, ISSN: 0001-0782. DOI: 10.1145/359545.359563.

[4] Paul E McKenney and Dipankar Sarma, "Towards hard realtime response from the Linux kernel on SMP hardware", in *Linux.conf.au*, Canberra, Australia, Apr. 23, 2005.

[5] Daniel Pierre Bovet and Marco Cesati, *Understanding the LINUX Kernel: From I/O Ports to Process Management*, 1st Edition. O'Reilly Media, Oct. 2000, ISBN: 0-596-00002-2.

[6] Arnd C Heursch, Dirk Grambow, Alexander Horstkotte, and Helmut Rzehak, "Steps towards a fully preemptable Linux kernel", in *Real-Time Programming 2003: Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming (WRTP'03)*, Łagów, Poland: Published for the International Federation of Automatic Control by Elsevier, May 14–17, 2003, ISBN: 0-080-44203-X.

[7] Sreekrishnan Venkateswaran, *Essential Linux Device Drivers*, 1st Edition. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008, ISBN: 9780132396554.

[8] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux Device Drivers*, 3rd Edition. O'Reilly Media, Inc., 2005, ISBN: 0-596-00590-3.

[9] Jonathan Corbet, "(Nearly) full tickless operation in 3.10", May 8, 2013. URL: http://lwn.net/Articles/549580/ (visited on 03/18/2014).

[10] Robert Love, *Linux Kernel Development*, 2nd Edition. Indianapolis, Ind: Novell Press, Jan. 2005, ISBN: 0-672-32720-1.

[11] Thomas Gleixner and Douglas Niehaus, "Hrtimers and beyond: Transforming the linux time subsystems", in *Proceedings of the Linux symposium*, vol. 1, Ottawa, Canada, Jul. 2006, pp. 333–346.

[12] Jonathan Corbet, "The high-resolution timer API", Jan. 16, 2006. URL: https://lwn.net/Articles/167897/ (visited on 03/19/2014).

[13] Lennart Poettering, Pierre Ossman, Shahms E King, *et al.*, "The PulseAudio Sound Server", Presented at the linux.conf.au 2007 conference, University of New South Wales, Sydney, Australia, Jun. 17, 2007. URL: http://www.linux.org.au/conf/2007/talk/211.html.

[14] Lennart Poettering, "Cleaning up the linux desktop audio mess", in *Proceedings of the Linux Symposium*, vol. 2, Ottawa, Canada, Jun. 2007, pp. 145–150.

[15] Atmel Corporation, "ATmega48A/PA/88A/PA/168A/PA/328/P [DATASHEET]", Feb. 12, 2013. URL: http://www.atmel.com/Images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf (visited on 06/29/2014).

[16] Gustavo de Veciana, "Leaky buckets and optimal self-tuning rate control", in *Global Telecommunications Conference, 1994. GLOBECOM '94. Communications: The Global Bridge., IEEE*, IEEE, vol. 2, Nov. 1994, pp. 1207–1211. DOI: 10.1109/GLOCOM.1994.512849.

[17] alsa-devel, portaudio, and audacity-devel mailing lists, "Questions about virtual ALSA driver (dummy), PortAudio and full-duplex", Jul. 24, 2013. URL: http://thread.gmane.org/gmane.linux.alsa.devel/110686 (visited on 04/01/2014).

[18] xujianqun, "音频框架结构 (Audio frame structure)", Nov. 8, 2011. URL: http://blog.csdn.net/xujianqun/article/details/6949078 (visited on 04/05/2014).

[19] Takashi Iwai, "ALSA Sequencer System", Presented at the 7th International Linux Kongress, der Friedrich-Alexander Universität Erlangen-Nürnberg, Germany, Sep. 21, 2000. URL: http://www.alsa-project.org/~tiwai/lk2k/lk2k.html (visited on 04/05/2014).

[20] ——, "Sound Systems on Linux: From the Past To the Future", in *UKUUG Linux 2003 Conference*, George Watson's College, Edinburgh, Scotland, Aug. 1, 2003.

[21] Olivier Hersent, Jean-Pierre Petit, and David Gurle, *Beyond VoIP Protocols: Understanding Voice Technology and Networking Techniques for IP Telephony*. Wiley, 2005, ISBN: 0-470-02362-7.

[22] Roger L. Freeman, *Telecommunication System Engineering*, ser. Wiley Series in Telecommunications and Signal Processing. Wiley, 2004, ISBN: 0-471-45133-9.

[23] Bill Waggener, *Pulse Code Modulation Techniques*. Thomson Publishing, 1995, ISBN: 0-442-01436-8.

[24] Marina Bosi and Richard E. Goldberg, *Introduction to Digital Audio Coding and Standards*. Kluwer Academic Publishers, Dec. 2002, ISBN: 1-4020-7357-7.

[25] Eric S. Raymond, *The Art of UNIX Programming*, ser. Addison-Wesley professional computing series. Pearson Education, 2003, ISBN: 0-132-46588-4.

[26] Valentijn Sessink, "Alsa-sound-mini-HOWTO", Nov. 12, 1999. URL: http://www.tldp.org/HOWTO/Alsa-sound.html (visited on 04/14/2014).

[27] alsa-devel mailing list, "Problems with a PCM driver", Aug. 31, 2003. URL: http://thread.gmane.org/gmane.linux.alsa.devel/8634/focus=8643 (visited on 04/16/2014).

[28] alsa-project.org, "ALSA project - the C library reference: PCM (digital audio) interface". URL: http://www.alsa-project.org/alsa-doc/alsa-lib/pcm.html (visited on 04/16/2014).

[29] Andreas Trøllund Boye, Carsten Steffensen, Gustav Høgh, Jes Toft Kristensen, Niels Christian Holm, and Per Kingo Jensen, "PC FM-radio receiver", Electrical and Electronic Engineering, The Faculty of Engineering and Science, Aalborg University, 5th semester project report, Dec. 13, 2005. URL: http://kom.aau.dk/group/05gr506/report/node21.html (visited on 04/18/2014).

[30] Aquiles Yáñez Cañas, "ALSA API - Sample Programs With Source Code By Aquiles Yanez", (dead link, unarchived). URL: http://alumnos.elo.utfsm.cl/~yanez/alsa-sample-programs/ (visited on 08/09/2013).

[31] alsa-project.org, "FramesPeriods - AlsaProject", Nov. 5, 2010. URL: http://www.alsa-project.org/main/index.php/FramesPeriods (visited on 04/18/2014).

[32] alsa.opensrc.org, "Frame", Aug. 10, 2013. URL: http://alsa.opensrc.org/Frame (visited on 04/17/2014).

[33] Jan Newmarch, "Programming and Using Linux Sound", Mar. 27, 2014. URL: http://jan.newmarch.name/LinuxSound/index.html (visited on 04/17/2014).

[34] Lennart Poettering, "What's Cooking in PulseAudio's glitch-free Branch", Apr. 8, 2008. URL: http://0pointer.de/blog/projects/pulse-glitch-free.html (visited on 04/17/2014).

[35] John Cowley, *Communications and Networking: An Introduction*, ser. Undergraduate Topics in Computer Science. Springer London, 2012, ISBN: 1-447-14356-6.

[36] Ulrich Drepper, "What every programmer should know about memory", Nov. 21, 2007. URL: http://people.redhat.com/drepper/cpumemory.pdf (visited on 05/02/2014).

[37] Atul P. Godse and Deepali A. Godse, *Microprocessors And Interfacing*, 1st Edition. Pune, India: Technical Publications, 2009, ISBN: 81-8431-125-7.

[38] Karlston D'Emanuele, "DMA Controller". URL: http://members.tripod.com/~Eagle_Planet/dma_controller.html (visited on 05/02/2014).

[39] Frank Durda IV, "DMA: What it Is and How it Works", *FreeBSD Handbook*, Jul. 7, 1998. URL: http://www.pl.freebsd.org/handbook/handbook320.html (visited on 05/03/2014).

[40] Chatchai Jantaraprim, "An asynchronous DMA controller", Master's thesis, University of Manchester, United Kingdom, Jan. 1999.

[41] Intel Corporation, "High Definition Audio Specification", Jun. 17, 2010. URL: http://www.intel.com/content/www/us/en/standards/high-definition-audio-specification.html (visited on 05/19/2014).

[42] Realtek Semiconductor Corp., "Datasheets (Computer Peripheral ICs > PC Audio Codecs > High Definition Audio Codecs > 2-Channel)". URL: http://www.realtek.com.tw/downloads/downloadsView.aspx?Langid=1&PFid=27&Level=5&Conn=4&ProdID=166&DownTypeID=1&GetDown=false&Downloads=true (visited on 05/19/2014).

[43] alsa-project.org, "Test latency.c - AlsaProject", Sep. 18, 2013. URL: http://www.alsa-project.org/main/index.php/Test_latency.c (visited on 05/24/2014).

[44] Intel Corporation, “High Definition Audio Energy Efficient Buffering: Spec”, Apr. 27, 2011. URL: http://www.intel.com/content/www/us/en/chipsets/high-definition-audio-energy-efficient-audio-buffering.html (visited on 06/09/2014).

[45] Alan Horstmann, “PortAudio - SVN Commit 1897: Alsa: Fix handling of poll descriptors in PaAlsaStream_WaitForFrames()”, Aug. 13, 2013. URL: https://www.assembla.com/code/portaudio/subversion/commit/1897 (visited on 06/09/2014).

[46] Compaq Computer Corporation, Hewlett-Packard Company, Intel Corporation, Lucent Technologies, Inc., Microsoft Corporation, NEC Corporation, and Koninklijke Philips N.V., “Universal Serial Bus Revision 2.0 specification (usb_20.pdf)”, Mar. 11, 2014. URL: http://www.usb.org/developers/docs/usb20_docs/usb_20_042814.zip (visited on 06/23/2014).

[47] Jay Senior, “USB Client Driver Etiquette”, Microsoft Corporation, presentation, May 10, 2001. URL: http://www.usb.org/developers/presentations/pres0501/Senior_Driver_Etiq_Final.ppt (visited on 06/28/2014).

[48] sigrok Wiki, “Saleae Logic - sigrok”, Jul. 28, 2013. URL: http://sigrok.org/wiki/Saleae_Logic (visited on 06/13/2014).

[49] Future Technology Devices International Limited, “FT232R USB UART IC Datasheet”, Mar. 2012. URL: http://www.ftdichip.com/Documents/DataSheets/ICs/DS_FT232R.pdf (visited on 06/15/2014).

[50] arduino.cc, “Arduino Duemilanove (2009) Schematic”, Oct. 17, 2008. URL: http://arduino.cc/en/uploads/Main/arduino-duemilanove-schematic.pdf (visited on 06/20/2014).

[51] Future Technology Devices International Limited, “AN232B-04 Data Throughput, Latency and Handshaking”, Feb. 2006. URL: http://www.ftdichip.com/Documents/AppNotes/AN232B-04_DataLatencyFlow.pdf (visited on 06/23/2014).

[52] linux-usb mailing list, “FTDI USB-to-UART converters and tcdrain()”, Oct. 2, 2012. URL: http://thread.gmane.org/gmane.linux.usb.general/72004 (visited on 07/01/2014).

[53] ——, “Using both usbmon and ftrace?”, Dec. 25, 2013. URL: http://comments.gmane.org/gmane.linux.usb.general/100774 (visited on 07/04/2014).

[54] Ubuntu: “kst” package: Bugs, “Bug #1258878: Weird interpolation for vectors in different time base”, Dec. 8, 2013. URL: https://bugs.launchpad.net/ubuntu/+source/kst/+bug/1258878 (visited on 07/07/2014).

[55] Jonathan Corbet, “On DTrace envy”, Aug. 7, 2007. URL: http://lwn.net/Articles/244536/ (visited on 07/10/2014).

[56] alsa-project.org, “XRUN Debug - AlsaProject”, Jul. 15, 2010. URL: http://www.alsa-project.org/main/index.php/XRUN_Debug (visited on 07/10/2014).

E-publication notice

Please note that the following papers from part II (original page range 159-232):

[II-D] "An analog I/O interface board for Audio Arduino open soundcard system"

[II-E] "Towards an open sound card — a bare-bones FPGA board in context of PC-based digital audio"

[II-F] "Open soundcard as a platform for practical, laboratory study of digital audio: a proposal"

... as well as the papers from part III (original page range 339-386), have been left out from this online version. Please consult the original publication outlets.

ISSN (online): 2246-1248
ISBN (online): 978-87-7112-311-1

AALBORG UNIVERSITY PRESS